TextTransformer 1.7.5

Inhaltsverzeichnis

Teil I

1 zu dieser Hilfe

Entsprechend den vielfältigen Anwendungsmöglichkeiten des TextTransformers ist das Programm für weite Anwenderkreise gedacht. Diese Hilfe wurde daher so verfasst, dass sie auch von Programmierneulingen nachvollziehbar sein sollte.

Die Beispiele sind wie ein **Tutorium** aufgebaut und geben einen guten Einblick in die Möglichkeiten des TextTransformers. Es ist empfehlenswert, mit diesen Beispielen zu experimentieren, bevor man eigene Projekte entwickelt.

Die Einführung gibt einen kurzen Überblick über die Arbeitsweise des TextTransformers und erklärt einige grundlegende Begriffe.

Ein Assistent hilft bei der Erstellung neuer Projekte. Das Spielen mit den Optionen dieses Assistenten, eignet sich zum ersten Kennenlernen des TextTransformers.

Der Ausdruck **TETRA** wird im folgenden als abkürzende Bezeichnung des TextTransformer-Programms verwendet.

TETRA ist ein Programm zum Erstellen, Testen und Ausführen der Regeln und Anweisungen, die einen Quelltext in einen Zieltext transformieren.

TETRA gibt es in einer freien, in einer Standard- und einer Professional-Version. Die Standard-Version ist gegenüber der freien um Index-Operationen erweitert. Diese umfassen u.a. die Möglichkeit des Zugriffs auf Unterausdrücke der die Token definierenden regulären Ausdrücke und die Verwendung von Containerklassen. Weiterhin stehen der Transformationsmanager, der Variablen-Inspektor und die meisten Assistenten nur ab der Standard-Version zur Verfügung. Mit der Professional-Version können zudem C++-Parserklassen erzeugt werden.

In **blauer Schrift** werden Abschnitte dieser Hilfe hervorgehoben, die nur die Professional-Version betreffen.

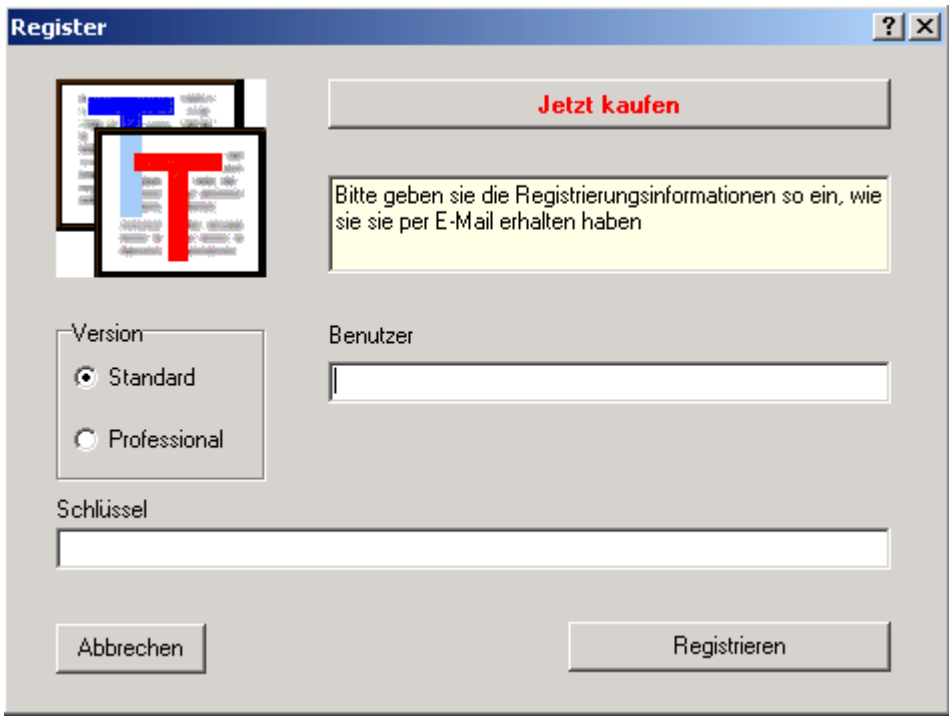
Die **rote Schrift** dient der Hervorhebung von Warnhinweisen oder anderen wichtigen Anmerkungen.

TextTransformer

Teil II

2 Registrierung

Der TextTransformer wird ausschließlich über das Internet vertrieben. Es gibt keine CD und kein gesondertes Handbuch.
Die **Registrierung** des TextTransformers, d.h. das Freischalten der Funktionen der Standard- oder Professional-Version, wird über den Menüpunkt *Registrierung,* der sich im Hauptmenü unter "Hilfe" befindet gestartet. Dort wird folgende Dialogbox geöffnet:



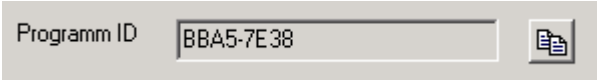
Zur Registrierung der **Standard Version** müssen sie neben ihren Adressangaben und den Angaben zur Zahlungsweise einen **Benutzernamen** (aus mindestens acht Zeichen) übermitteln. Zur Registrierung der **Professional Version** ist zusätzlich eine **Programm-ID** erforderlich (s.u.).

Formulare für die entsprechenden Eingaben werden in ihrem Internet-Browser angezeigt, wenn sie on-line sind und den **Jetzt kaufen** Schalter betätigen.

Nach erfolgter Prüfung ihrer Kreditkarte wird ihnen automatisch eine **E-Mail** zugeschickt, die die Registrierungsdaten (Benutzername und Schlüssel) enthält. Diese müssen nun in die entsprechenden Felder der oben gezeigten Dialogbox übertragen werden. Zunächst wird durch Anklicken des entsprechenden Knopfes festgelegt, ob eine Registrierung der **Standard- oder der Professional-Version** erfolgen soll. Dann kopieren sie bitte **Benutzername** und **Schlüssel** unverändert aus der E-Mail in die entsprechenden Eingabefelder der Dialogbox.
Nach Betätigung des **Schalters *Registrieren*** wird die Dialogbox automatisch geschlossen und eine Meldung erscheint, die den Erfolg der Registrierung bestätigt.

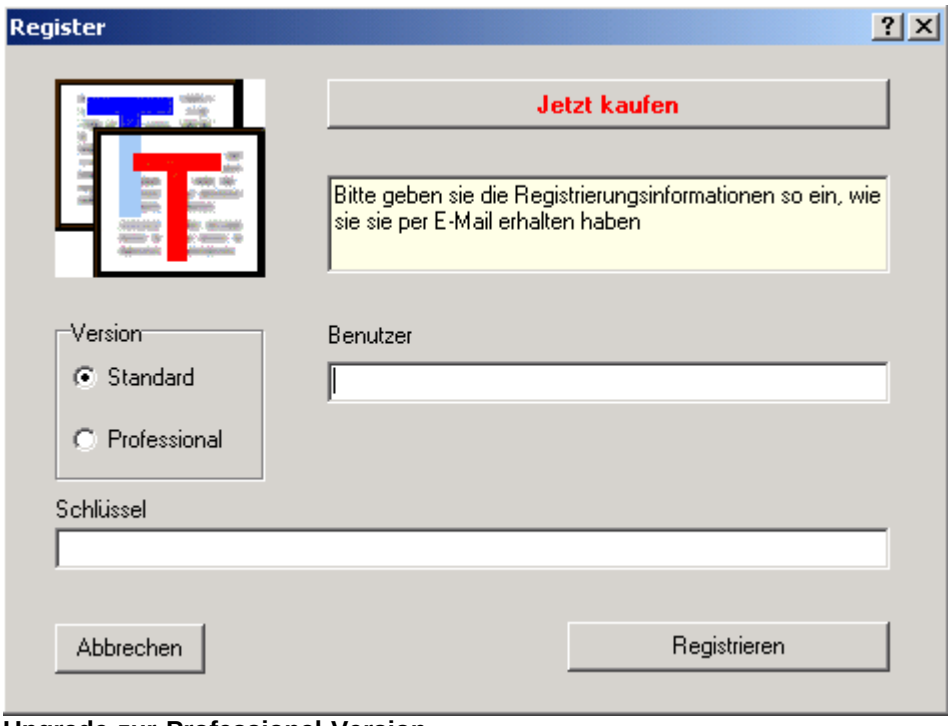
Ermittlung der Programm-ID zur Registrierung der Professional Version

Die Programm-ID, die zur Registrierung der Professional-Version erforderlich ist, wird ihnen angezeigt, sobald sie in der Dialogbox den Knopf *Professional* anklicken. Es erscheint ein zusätzliches grau hinterlegtes Feld mit einer Zahlen- und Buchstabenkombination.



Diese Programm-ID wird in die Zwischenablage kopiert, wenn sie den rechts liegenden Schalter betätigen.
Die Programm-ID ist spezifisch für ihre Hardware-Konfiguration. Die registrierte Professional Version lässt sich nur auf dem Computer ausführen, auf dem sie ursprünglich installiert wurde.

Falls sie die Professional Version des TextTransformer auf einem anderen Computer installieren wollen als dem, aud dem sie das Programm ursprünglich heruntergeladen haben, so sollten sie es unverzüglich auf ein transportables Speichermedium kopieren und *nicht* auf dem ersten Computer registrieren.
(Für die Standard Version sind keine besonderen Vorkehrungen erforderlich. Sie lässt sich beliebig transferieren.)



Upgrade zur Professional Version

Falls sie die Standard Version des TextTransformers registriert haben, erscheint in der Dialogbox anstelle des **Jetzt kaufen** Schalters ein Schalter, der ihnen die Möglichkeit bietet ihre Lizenz auf die Professional Version zu erweitern.

Teil III

3 Die wichtigsten Bedienungselemente

Bevor man beginnt eigene Projekte zu entwickeln, ist es empfehlenswert die Einführung zu lesen und mit dem Assistenten für neue Projekte und den Beispielen zu experimentieren.

1.
Entweder über das Datei-Menu ein Projekt und einen Text öffnen oder einen Text direkt in das Eingabefenster schreiben und auf der Produktionen-Seite mit

 eine neue Produktion erstellen und mit
 bestätigen.

2.
Ein Programm kann nur geparst und ausgeführt werden, wenn in der Auswahlbox der Werkzeugleiste

eine Startregel gesetzt ist, und nur, wenn die Aktionen aktiviert werden, wird auch eine Ausgabe erzeugt..

3.
Zum Parsen einer Produktion wird der Schalter:

Es gibt diesen Schalter sowohl auf der Werkzeugleiste, wo seine Betätigung die Startregel des Projekts parst, als auch auf der Produktionen-Seite, wo er die aktuelle Produktion als Startregel behandelt.

4.
Die Ausführung eines TextTransformer-Programms erfolgt entweder mit

 im langsameren Debugmodus

mit Haltemöglichkeiten und vollständigen Fehlermeldungen oder mit

 schnell im Ausführungsmodus

5.
Die Rückkehr aus dem Debug- bzw. Ausführungsmodus in den Bearbeiten-Modus erfolgt mit

 Zurücksetzen.

Teil IV

4 Die wichtigsten Syntaxelemente

Parser und Scanner

Gruppierung	(...)
Alternative	|
Option	?
Optionale Wiederholung	*
Wiederholung	+

Argumente werden an Produktionen oder Token in eckigen Klammern übergeben, die dem Namen der Produktion bzw. des Tokens folgen. Z.B. Name[iCount]

Interpreter

Die Syntax des Interpreters ist einfache C++-Syntax.

Tip.: Schreiben sie einfachen Code und verwenden sie besser zwei Anweisungen, als eine.

C++-Code wird innerhalb spezieller Klammern in die Parserbeschreibung eingeschoben:

nur im Interpreter ausführbar	{- ... -}
nur für C++-Export	{_ ... _}
für Interpreter und Export	{= ... =}
entsprechend Projektoptionseinstellung	{{ ... }}

Sehr häufig verwendete Anweisungen bzw, Ausdrücke sind:

In die Ausgabe schreiben	out << value;
zuletzt erkannter Text	State.str()
davor ausgelassenen Zeichen	State.str(-1)
zuletzt erkannter Text inklusive davor ausgelassenen Zeichen	State.copy()

Teil V

5 Wie beginnt man ein neues Projekt?

Bevor man beginnt eigene Projekte zu entwickeln, ist es empfehlenswert die Einführung zu lesen und mit dem Assistenten für neue Projekte und den Beispielen zu experimentieren.

Für viele Programmiersprachen und Formate kann man im Internet fertige Grammatiken finden. Falls eine solche Beschreibung existiert kann sie oft relativ einfach in die Syntax des TextTransformers übersetzt werden. Ein halb-automatischer Übersetzer für Coco/R-Grammatiken gehört zu den Beispielen des TextTransformer Pakets. In den Beispielen: E-Mail-Adresse und XML, wird demonstriert, wie vorhandene Syntax-Spezifikationen in TextTransformer-Programme überführt werden können.
Falls keine Syntax-Beschreibung existiert, muss sie selbst entwickelt werden. Hierfür gibt es Regeln und Erfahrungssätze, die als Leitfaden dienen können.

1. Setzen sie die erforderlichen Projekt-Optionen!

Z.B. ist es sehr wichtig, gleich zu Anfang der Entwicklung eines neuen Projekts die Zeichen auszuwählen, die für das Parsen der Texte keine Bedeutung haben. Standardmäßig sind u.a. die Zeilenumbruchszeichen als aus zulassende Zeichen gesetzt. Soll aber ein Parser entwickelt werden, der Zeilenenden erkennt, so muss diese Einstellung geändert werden.
Eine andere wichtige Entscheidung ist, ob alle literalen Token getestet werden sollen oder nicht. Faustregel: Alle Literale sollten getestet werden, falls eine formalisierte Sprachen mit definierten Schlüsselworten, die an ausgezeichneten Positionen stehen, geparst werden soll. Andernfalls sollten nur die erwarteten Literale getestet werden. Nötigenfalls können auch die lokalen Optionen jeweils angepasst werden.

2. Konstruieren sie zunächst den Parser ohne semantische Aktionen!

Bei der Konstruktion des Parsers wird es immer wieder nötig oder angebracht sein Produktionen um zustellen und komplexe Produktionen durch Definition von Unter-Produktionen übersichtlicher zu machen. Wenn der Parser schon semantischen Code enthielte, müsste dieser bei jeder dieser Änderungen neu angepasst werden.

3. Entwickeln sie von "oben" nach "unten"!

Beginnen sie mit der allgemeinsten Produktion - die Startregel, die den gesamten Text erkennen soll - und zerlegen sie die Startregel dann zunächst in Unter-Produktionen, die Hauptbestandteile des Textes erkennen sollen. Nach dem gleichen Prinzip werden dann die Unterproduktionen weiter verfeinert. Soll z.B. ein Buch geparst werden, so würde die Startregel zunächst lauten:

```
Buch ::= SKIP  // damit wird der gesamte Text erfasst
```

Nach der ersten Verfeinerung:

```
Buch ::= SKIP? Kapitel+

Kapitel ::= TITEL SKIP
```

TITEL steht hier für einen regulären Ausdruck, der eine Kapitelüberschrift eindeutig von anderen

Textbestandteilen abhebt.
Anmerkung: Ein solcher Ausdruck existiert gewiss nicht für alle Bücher. Das Buch wird hier als Beispiel für eine Textstruktur verwendet, die jeder kennt. Der Buch-Parser funktioniert nur für syntaktisch ideale Bücher. (Z.B. TITLE ::= \d\.[^\r\n]+ // falls der Text dieser Seite als "Buch" genommen wird.)

Nun kann die Chapter-Produktion weiter verfeinert werden:

Kapitel ::= TITEL Absatz+

Absatz ::= EOL+ SKIP

EOL ::= \r?\n // Zeilenende

Der Vorteil dieser Vorgehensweise von "oben" nach "unten" ist, dass in jedem Stadium der Entwicklung der aktuelle Parser an allen "Büchern" getestet werden kann. Eventuelle Fehler können so schon in einem frühen Entwicklungsstadium entdeckt werden.

Hinweis: Mit dem Transformations-Manager können viele Beispiele in einem Rutsch getestet werden. Schlägt ein solcher Test fehl, kann der entsprechende Text mit einem Klick in der IDE geöffnet werden.

4. Wählen sie die Art der Transformation

Im Prinzip gibt es drei Weisen, wie der Parser zu einem vollständigen Konvertierungsprogramm ergänzt werden kann. Sie unterscheiden sich darin, was jeweils mit den erkannten Textabschnitten getan wird.

a) Textabschnitte werden unmittelbar bearbeitet und in die Ausgabe geschrieben.

b) die Textabschnitte werden in Variablen geschrieben, die als (Referenz-)Parameter an andere Produktionen zurück- oder weitergegeben werden und dort ausgewertet und kombiniert werden können.

c) es wird ein Parse-Baum erzeugt und die Verarbeitung der Textabschnitte erfolgt erst nachdem der gesamte Text geparst wurde.

Die letzte Methode ist die variabelste ist, da im Prinzip noch auf sämtliche Textabschnitte zugegriffen werden kann, und da mit dem Parse-Baum je nach Funktionstabelle verschiedene Ausgaben erzeugt werden können. Soll ein Übersetzer entwickelt werden, der ein Eingabeformat in mehrere Ausgabeformate konvertieren soll, so ist die Verwendung eines Parse-Baums nahezu unverzichtbar. Die Entwicklung eines solchen Übersetzers ist jedoch ungleich schwieriger, als die direkte Verarbeitung des Quelltextes mit einer der beiden anderen Methoden.

Ist die Reihenfolge in der die bearbeiteten Textabschnitte ausgegeben werden sollen in etwa mit der Folge identisch, in der sie erkannt wurden, dann ist die erste Methode der direkten Textausgabe zu empfehlen. Müssen die erkannten Textteile neu angeordnet werden oder hängt die Bearbeitung eines Teils von einem Text ab, der erst später ermittelt wird, so ist die zweite Methode zu empfehlen.

Wenn über die Art der Transformation entschieden wurde, können verschiedene Assistenten helfen, Parameter, Variablen-Deklarationen oder Baumknoten in Produktionen einzusetzen.

5. Erstellen sie ein Kopierprogramm, bevor sie den endgültigen Code für die Transformation schreiben!

Diese Regel gilt nur für Projekte, bei denen der Quelltext an einigen ausgezeichneten Stellen modifiziert werden soll. Indem zunächst ein Programm erstellt wird, dass den Quelltext lediglich kopiert, kann durch Vergleich mit dem Zieltext leicht ermittelt werden, ob die Ausgabe vollständig ist.

Wie beginnt man praktisch

5.1 Praxis

1. Neues Projekt erstellen

Im Datei-Menü ist der Eintrag Neues Projekt zu wählen. Zunächst erscheint der Assistent für neue Projekte.

2. Namen vergeben

Auf der ersten Seite des Assistenten sind ein Projekt-Name und ein Name für die Startregel einzugeben. Meist ist es sinnvoll beide gleich zu nennen, in unserem Beispiel: Buch. Wird dieser Name in das Feld für den Projekt-Namen eingegeben, so erscheint er auch unmittelbar im Feld für die Startregel und im linken Menü des Assistenten wird der nächste Schritt angezeigt: Projekt-Typ.

3. Projekt-Typ wählen

Auf die Seite für den Projekt-Typ kann nun gewechselt werden indem man entweder den Schalter Next anklickt oder indem man auf den Menü-Eintrag klickt.
Auf dieser Seite gibt es vier verschiedene Auswahlmöglichkeiten für Projekt-Typen. Der letzte: "Neues Projekt ohne Vorgaben" soll hier gewählt werden. Sobald er angeklickt wird, wird automatisch auf die nächste Seite "Fertig stellen" gewechselt, wo die gewählten Namen nochmals angezeigt werden.

4. Roh-Projekt speichern

Sobald der *Finish*-Schalter betätigt wird, erscheint ein Dateiauswahl-Dialog, wo der Ordner ausgewählt wird, in dem das Projekt gespeichert werden soll.

5. Startregel editieren

Wenn das Projekt gespeichert wurde, wird der Assistent automatisch geschlossen und man befindet sich auf der Hauptseite der Tetra-IDE. In der Auswahlbox für die Startregel in der Werkzeugleiste wird der gewählte Name *Buch* bereits angezeigt. Ebenso ist er in den Projekt-Optionenbereits eingetragen und er ist auch im Syntaxbaum auf der rechten Seite der IDE zu sehen. Wird er dort angeklickt, so öffnet sich die Tabellen-Seite *Produktionen* und man sieht die Definition der *Buch*-Produktion: SKIP. Mit SKIP wird jeder Text erkannt. Die Definition muss also bearbeitet werden.

Wie im im theoretischen Teil unter Punkt 3 vorgeschlagen, wird der Definitionstext der Buch-Produktion nun ersetzt durch

```
SKIP? Kapitel+
```

Der Ausdruck *Kapitel* wird zunächst in normaler Schriftart dargestellt. Wäre die Kapitel-Produktion bereits definiert, würde sie in brauner Fettschrift angezeigt.

6. Kapitel-Produktion einfügen

Diese Definition soll jetzt erfolgen. Dazu wird zunächst mit dem Plus-Schalter eine neue Regel erzeugt die den Namen "Kapitel" erhält. Als Definitionstext wird hier

```
TITLE SKIP
```

geschrieben. Nach Bestätigung ist diesmal der Ausdruck *TITLE* in normaler Schrift dargestellt. Geht man mit dem Zurück-Schalter auf die *Buch*-Produktion zurück, sieht man dass dort *Kapitel* mittlerweile hervorgehoben ist.
Nun muss also *TITLE* definiert werden. Die Großschreibung des Wortes soll zum Ausdruck bringen, dass es sich nicht um eine Produktion, sondern um ein Token handelt. Eine solche Großschreibung ist aber nicht notwendig.

7. TITLE-Token einfügen

Token werden auf der Token-Seite des TextTransformer definiert. Die Bedienung ist hier analog wie auf der Produktionen-Seite. Mit dem Plus-Schalter wird ein neues Token erzeugt, das den Namen *TITLE* erhält. Ein Titel sei definiert durch

```
\d\.[^\r\n]+
```

Mit diesem Ausdruck werden Überschriften erkannt, die mit einer einzelnen Ziffer beginnen auf die ein Punkt und eine beliebige Folge von Zeichen bis zum Zeilenende folgt. Das ist sicher keine allgemein gültige syntaktische Definition für Titel; sie dient nur als Beispiel.

8. Projekt kompilieren

Nun ist das Projekt vollständig und kann kompiliert werden, indem in der Haupt-Werkzeugleiste auf den Schalter *Startregel parsen* geklickt wird. Im Syntaxbaum kann man sich jetzt auch die Strukturen der Produktionen ansehen.

9. Erneut speichern

Bitte vergessen Sie nicht, das Projekt zu speichern. Sie können es jetzt verfeinern, so wie im theoretischen Teil vorgeschlagen.

TextTransformer

Teil VI

6 Einführung

TETRA ist ein Akronym von **Text-Transformer**.. TETRA dient dazu Texte umzuwandeln, d.h. Quell- in Zieltexte zu übersetzen.

Was dies genauer bedeutet, kann am besten durch Beispiele erklärt werden.
Diese Umwandlung kann aus einfachen **Wortersetzungen** bestehen, z.B. der Ersetzung von "TextTransformer" durch "TETRA". Im Unterschied zu einer normalen Textverarbeitung, kann man mit dem TextTransformer ganze Listen von Wortpaaren in einem Zug abarbeiten.
Ein wenig anspruchsvoller wären schon **Wortumstellungen**. So könnten Adresslisten, die aus Name, Adresse und Telefonnummer bestehen in die Form transformiert werden: Telefonnummer, Name, Adresse.
Ein anderes Beispiel wäre die **Extraktion bestimmter Daten** aus einem Text. Z.B. könnten Namen und Preise bestimmter Produkte aus einem Katalog extrahiert und zu einer Liste zusammengestellt werden.
Auch ist es möglich die extrahierten Daten bestimmten Aktionen zu unterziehen, z.B. **Berechnungen**. Preise aus einer Rechnung könnten extrahiert und addiert werden. In diesem Fall bestünde die Textumwandlung darin, aus einer Rechnung einen einzelnen Summenausdruck zu bilden. Daran sieht man, das der Ausdruck Textumwandlung sehr allgemein zu verstehen ist.

Die bisherigen Beispiele waren recht einfacher Natur. Mit etwas Übung lassen sich derartige Transformationsprogramme **in wenigen Minuten** schreiben. Wer noch nicht so geübt ist, der braucht zwar etwas länger, aber die Zeit vergeht wie im Fluge, da die Entwicklung eines Programms mit dem TextTransformer ein **spielerisches Vergnügen** ist: auf eventuelle Fehler weist der TextTransformer sofort hin und man kann alles einfach mal eben Stück für Stück ausprobieren.

Seine ganze Stärke zeigt der TextTransformer, wenn es darum geht komplexe Grammatiken zu verarbeiten, z.B. bei der **Übersetzung einer Programmiersprache in eine andere**.
Schließlich kann TETRA auch auf eine ganz andere Weise verwendet werden, nämlich nicht zur Analyse von Texten vorgegebener Struktur, sondern zur Entwicklung solcher Strukturen selbst. Mit TETRA können **neue Programmiersprachen entwickelt** werden. So sind z.B. die zentralen Teile des TETRA-Programms mit sich selbst erzeugt worden.

6.1 Wie funktioniert der TextTransformer?

Bei allen von TETRA durchzuführenden Texttransformationen sind zwei Hauptaufgaben zu erledigen

1. die **Analyse** des Quelltextes
2. die **Synthese** des Zieltextes

Ein Quelltext wird gemäß seiner syntaktischen Form zerlegt und sogleich als Zieltext in einer neuen Form wieder aufgebaut. Vom Programmbenutzer müssen hierfür sowohl die Syntaxregeln des Quelltextes als auch die Anweisungen zur Synthese des Zieltextes formuliert werden.

6.2 Analyse

Die Analyse des Quelltextes erfolgt in zwei Schritten

Bei der **lexikalischen Analyse** wird der Quelltext in Worte und Satzzeichen etc. zerlegt. Allgemeiner gesagt: die lexikalische Analyse ist die Erkennung der sogenannte **Token**. Die Token, auch **Terminalsymbole** genannt, bestehen aus einem oder mehreren aufeinander folgenden Zeichen. Diese Aufeinanderfolge kann jeweils als ein spezielles Zeichenmuster aufgefasst werden und als solche lassen sie sich durch sogenannte **reguläre Ausdrücke** beschreiben.
Je nach Art des Textes können Token Verschiedenes sein. Bei mathematischen Texten kämen z.B. Namen, Zahlen und Operatoren in Betracht, bei umgangssprachlichen Texten bestimmte Worte, Wortgruppen, Sätze oder Wortteile, in Datendateien die einzelnen Felder eines Datensatzes.
Zugleich mit der lexikalischen Analyse werden bedeutungslose Zeichen beseitigt. Das können je nach Grammatik Leerzeichen und Tabulatoren, Leerzeilen, Kommentare etc. sein.

Bei der **syntaktischen Analyse** wird ermittelt in welcher Abfolge Token im Text auftreten. Diese Abfolge ist definiert durch Reihen oder Alternativen von Token, die im Text wiederholt aufeinander folgen. Z.B. kann ein Text einfach als eine Reihe von Textzeilen aufgefasst werden oder als das wiederholte Auftreten von Wortgruppen, die durch Satzzeichen voneinander getrennt sind. Der Text kann aber auch einer Grammatik gehorchen, die durch komplexere Regeln beschrieben wird.
Eine Syntaxregel wird eine **Produktion** genannt oder auch ein **Nonterminalsymbol**.

6.3 Synthese

Bei der Analyse des Textes wurde dieser in seine Bestandteile zerlegt: die Token selbst und bestimmte regelmäßige Folgen von Token.
Die Synthese setzt diese Bestandteile nun zu einem neuen Text zusammen oder führt mit den Bestandteilen bestimmte **semantische Aktionen** durch. Die Anweisungen der semantischen Aktionen werden in die Definition der Produktionen eingebettet.

Die Anweisungen zur Synthese sind der Programmiersprache C++ entnommen. Ein Teilsystem von C++ ist in TETRA als interpretierte Sprache integriert, d.h. eine bestimmte Menge von C++-Befehlen lässt sich unmittelbar in der TETRA-Programmierumgebung ausführen. (Im Unterschiede hierzu ist C++ normalerweise eine Sprache, die zunächst kompiliert werden muss, d.h. dass eine ausführbare Datei - ein "Programm" - hergestellt wird, die dann gesondert ausgeführt wird.) Die in TETRA integrierten C++-Anweisungen erlauben beispielsweise die Verkettung der Textbestandteile in neuer Reihenfolge oder die Ersetzung dieser Bestandteile durch andere und vieles mehr.

In der Professional-Version von TETRA kann Programmcode zur Einbindung in externe Anwendungen erzeugt werden, der die Möglichkeiten der Programmiersprache C++ uneingeschränkt nutzt.

6.4 Reguläre Ausdrücke

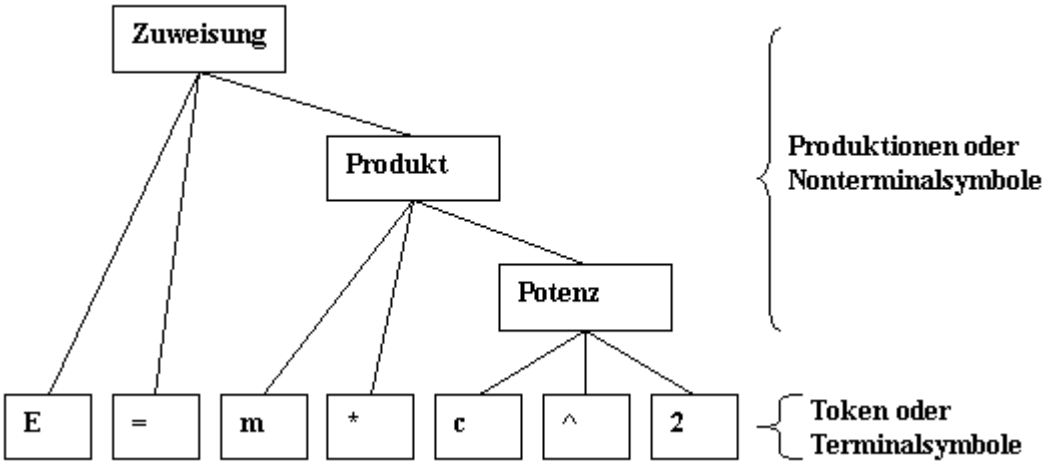
Es wurde bereits erwähnt, dass die Textbestandteile in die TETRA den Text zerlegt *Token* oder *Terminalsymbole* genannt werden. Diese werden anhand von *regulären Ausdrücken* identifiziert. Reguläre Ausdrücke sind bekannt aus vielen Skriptsprachen und Textverarbeitungen. Sie erweitern die Möglichkeiten zur Wortsuche und Ersetzung, indem sie es erlauben nicht nur einzelne bestimmte (Teil-)wörter zu suchen, sondern Gruppen von Worten, die ein gemeinsames Muster haben.

Reguläre Ausdrücke definieren anhand weniger einfacher Regeln Zeichenmuster, d.h. sie beschreiben auf welche Weise Buchstaben oder andere Zeichen aufeinander folgen.
Typische Textstücke (Token), die durch reguläre Ausdrücke identifiziert werden können sind Worte, Zahlen, Uhrzeiten und Datumsangaben, Anführungen, Sätze oder Formatierungszeichen.

6.5 Syntaxbaum

Ebenso wie die Terminalsymbole oder Token durch reguläre Ausdrücke beschrieben werden, kann auch Muster von Token selbst durch Regeln beschrieben werden. So besteht z.B. ein Hauptsatz aus einer Folge von Wörtern gefolgt von einem Punkt, oder eine Tabelle besteht aus dem Tabellenkopf mit den Namen für die einzelnen Spalten gefolgt von einer Reihe Zeilen..

Die Syntaxanalyse zeigt auf, in welchem grammatikalischen Zusammenhang die Token zueinander stehen. Dieser Zusammenhang kann als eine Baumstruktur, als eine sogenannter Syntaxbaum, dargestellt werden.



In diesem Beispiel sind die Token denkbar einfach. Sie bestehen aus jeweils nur einem Zeichen: "E", "m", "*", "c", "^", und "2". In der Abbildung sind die Token oder **Terminalsymbole** die Blätter des Baums und stehen in der untersten Zeile. Die Blätter sind grafisch dadurch ausgezeichnet, dass an ihnen jeweils nur eine Linie endet; grammatisch sind sie dadurch gekennzeichnet, dass sie sich nicht weiter zerlegen lassen (s. Anm.).
Dies unterscheidet sie von den anderen Knoten des Baums, die die sogenannten **Nonterminalsymbole** repräsentieren. Die Nonterminalsymbole lassen sich in die Terminalsymbole

zerlegen. Sie sind in der grafischen Darstellung Ausgangspunkt von Verzweigungen.

Im TextTransformer würde der Syntaxbaum der Abbildung in drei Baumstrukturen für die Nonterminalsymbole zerlegt in etwa folgendermaßen aussehen:

Anmerkung: Dass die Terminalsymbole sich grammatisch nicht weiter zerlegen lassen, heißt nicht, dass sie nicht in Zeichenfolgen zerlegbar wären, wie es bei komplexeren Token als in diesem Beispiel stets der Fall ist.

6.6 Produktionen oder Nonterminalsymbole

Die Nonterminalsymbole werden durch Regeln beschrieben, die die Abfolge der Terminalsymbole bestimmen. Eine solche Regel wird eine **Produktion** genannt. Zur Formulierung der Produktionen gibt es in TETRA eine Skriptsprache, deren Syntax, der der regulären Ausdrücke ähnelt. Im Unterschied zu den regulären Ausdrücken, deren Elemente die einzelnen Zeichen sind, beziehen sich die Operatoren (Verkettung, Wiederholung etc.) der TETRA-Skriptsprache auf Token, also auf Zeichenmuster. Insofern sind Produktionen gewissermaßen Muster von Mustern.

Eine Produktion erfüllt zwei Aufgaben. Neben der gerade erläuterten Aufgabe

1. zu bestimmen, was in einem Text erkannt wird
2. zu bestimmen, was mit dem Erkannten angefangen werden soll (Aktionen)

Punkt 2 bezeichnet die bereits oben erwähnte Synthese.

Entsprechend diesen beiden Aufgaben gibt es in einer TETRA-Produktion durch eine spezielle Klammerung voneinander abgehobene Abschnitte, die jeweils einer der beiden Aufgaben zugeordnet sind. Im Beispiel auf der folgenden Seite sind die Klammern"{{" und "}}" verwendet, um die mit Punkt 2 bezeichneten semantischen Aktionen vom syntaktischen Code (Punkt 1) zu trennen.

6.7 Produktionen als Funktionen

Der folgende Text ist für Benutzer gedacht, die schon über einige Programmiererfahrungen verfügen. Allen anderen wird das Nachvollziehen der Beispiele empfohlen.

Eine Produktion kann als eine Spezifikation zur Erzeugung einer Funktion angesehen werden, die einen Textabschnitt parst. Bei der Codeerzeugung entsteht aus dieser Spezifikation tatsächlich eine Funktion. Insofern können einer Produktion Parameter übergeben werden und sie selbst kann einen Wert zurück liefern. Innerhalb des Funktionskörpers der Produktion können Variablen und Konstanten definiert werden, die wiederum an andere Produktionen übergeben werden können, die wie Funktionen aufgerufen werden. Die aufgerufenen Produktionen parsen Unterabschnitte des Textabschnitts, den die aufrufende Produktion parst.

Beispiel

Im folgenden Beispiel ruft die *Outer*-Produktion die *Inner*-Produktion auf und erhält von letzterer einen String zurückgeliefert, der den Text enthält, den die *Inner*-Produktion erkannt hat.

```
Outer =
        "a"
        {{str s = }}
        Inner[s]
        {{out << "found a and " << s;}}


Inner =
        ( "b" | "c" )
        {{ return xState.str(); }}
```

Ist die Eingabe: "a b", dann
ist die Ausgabe: found: a and b

Ist die Eingabe: "a c", dann
ist die Ausgabe: found: a and c

Bei der Codeerzeugung erhält man (aufs Wesentliche verkürzt):

```
( typedef  std::string::const_iterator  cts;  )


// token ordered by symbol numbers
//    ------------------------------------------------
// Name                SymNo      regular expression
//    ------------------------------------------------
...
// a                      (  6 )         "a"
// b                      (  7 )         "b"
// c                      (  8 )         "c"


void COuterParser::Outer(cts xtBegin, cts xtEnd, plugin_type xpPlugin /*=NULL*/)
{
   sps xState(xtBegin, xtEnd);
   // ... falls xpPlugin == NULL neues lokale Plugin, sonst xpPlugin in xState einsetzen
  if(  m_apOuterScanner->GetNext(xState,  false))
  {
     Outer(xState, NULL, false);
  }
}
```

```
//-------------------------------------------------
void  COuterParser::Outer(sps&  xState,  ...)
{
    m_apT0_a_of_OuterScanner->GetNext(xState,  false);

  string s =
   Inner(xState,  ... );
  out << "found: a and " << s;
}

//-------------------------------------------------
std::string  COuterParser::Inner(sps&  xState,  ...)
{

  switch  (  Alt0_of_Inner(  xState.Sym()  )  )
  {
    case 7:  // "b"
      m_apT1_b_of_InnerScanner->GetNext(xState,  true);
    break;
    case 8:  // "c"
      m_apT2_c_of_InnerScanner->GetNext(xState,  true);
    break;
    default :
    throw  tetra::CTT_ErrorUnexpected(  ...);
  }

  return  xState.str();
}
```

Die Scanner enthalten jeweils die Information, welche Token zulässig sind und die Zustandsvariable xState enthält die Information darüber, welches Token erkannt wurde, und welches erwartet wird.

Anmerkung:
Die Klammern um "b" | "c" sind notwendig, da sonst bei der ersten Alternative ein undefinierter Wert zurückgegeben wird.

6.8 Vier Verwendungsweisen von Produktionen

Die Hauptaufgabe der Produktionen ist es, die Grammatik eines Textes so abzubilden, dass er sich damit parsen lässt. Im TextTransformer ist es aber auch möglich Produktionen für Aufgaben einzusetzen, die dieser Hauptaufgabe untergeordnet sind.

Insgesamt gibt es vier Verwendungsweisen für Produktionen in TETRA:

1. als Konstituenten des **Hauptparsers**
2. zur Vorausschau im Text
3. zum Parsen von Text-Einschlüssen
4. als Unterparser in semantischen Aktionen

6.9 Vorausschau

Die Entscheidung darüber mit welcher Produktion oder Verzweigung innerhalb einer Produktion die Analyse eines Textes fortgesetzt wird, häng stets davon ab, welche Token im Text folgen. Am effizientesten arbeitet ein Parser, wenn bereits das nächstfolgende Token diese Entscheidung ermöglicht. Wenn stets die Vorausschau um nur ein Token zum Analysieren des Textes ausreicht, so nennt man den Parser **LL(1)-konform**. Es ist eine Kunst des Entwicklers die Grammatik - die Menge der Produktionen - eines Textes so zu formulieren, dass der Parser LL(1)-konform wird.
Der TextTransformer bietet eine große Unterstützung bei dieser Aufgabe, da er automatisch Hinweise und Fehlermeldungen generiert, wenn die Grammatik nicht LL(1)-konform ist. **TETRA erlaubt jedoch auch die Vorausschau um beliebig viele Token**, wenn dies erforderlich sein sollte. Zur Formulierung einer solchen Vorausschau dienen erneut die bereits bekannten Produktionen. Eine Produktion kann versuchsweise auf einen Text angewendet werden, um zu testen, ob sie ihn parsen kann oder nicht. Abhängig vom Ergebnis dieses Tests kann die Analyse des Textes dann auf verschiedene Weise fortgesetzt werden. (Bei der versuchsweisen Anwendung von Produktionen werden keine semantischen Aktionen ausgeführt.)

Die Vorausschau sei nochmals an den konkreten Beispiel einer förmlichen Anrede am Anfang eines Briefes erläutert. Sie lautet entweder

```
Sehr geehrter Herr NAME
```

oder

```
Sehr geehrte Frau NAME
```

Um diese kurzen Texte zu parsen, könnte man zunächst auf die Idee kommen, folgende Produktionen zu formulieren (Das Zeichen "|" trennt Alternativen voneinander, bedeutet also "oder"):

```
Anrede ::= AnredeBeginn NAME

AnredeBeginn ::=   "Sehr" "geehrter" "Herr"
                 | "Sehr"  "geehrte" "Frau"
```

Bei dieser Formulierung ist eine Vorausschau um zwei Token erforderlich. Erst nachdem das Wort "Sehr" erkannt ist kann durch das folgende Wort entschieden werden, welche Alternative von *AnredeBeginn* zu wählen ist und es muss wieder vor das Wort "Sehr" zurückgegangen werden, um mit der Verarbeitung des Textes zu beginnen

Besser sind folgende Produktionen:

```
Anrede ::= "Sehr" Ehrung NAME
Ehrung ::   "geehrter" "Herr"
          | "geehrte" "Frau"
```

Bei dieser Formulierung entscheidet stets das nächst Wort eindeutig, wie in den Produktionen fortzufahren ist.
Erstaunlich viele Texte lassen sich nach diesem LL(1)-Prinzip parsen, wenn man die Regeln entsprechend entwirft.

Dennoch gibt es Fälle bei denen eine Vorausschau um nur ein Token prinzipiell nicht ausreicht. Im TextTransformer ist es möglich in solchen Fällen beliebig weit im Text voraus-zuschauen. Z.B. könnte es nötig sein schon vor dem Parsen eines Satzes zu wissen, ob es sich um einen Fragesatz handelt oder nicht. Der Fragesatz ist aber erst am Ende des Satzes durch das Fragezeichen zu identifizieren. Das ließe sich folgendermaßen bewerkstelligen:

```
IF( Fragesatz() )
  Fragesatz
ELSE
  Normalsatz
END

Fragesatz :: FragesatzWortfolge "?"
Normalsatz ::= NormalsatzWortfolge ( "." | "!" )
```

6.10 Einschlüsse / Kommentare

Typisches Beispiel für das, was hier mit einem Einschluß bezeichnet wird, sind Kommentare in Programmiersprachen. Der Begriff "Einschluss" wurde hier gewählt, um die allgemeine Struktur dieses Beispiels zu bezeichnen.

Kommentare oder andere Einschlüsse können an beliebigen Stellen in Texte eingeschoben sein. Die syntaktische Struktur der Texte wird durch die Einschlüsse selbst dann nicht zerstört, wenn in ihnen Schlüsselworte der Grammatik (Programmiersprache) verwendet werden.

Beispiel:.

In den Skripten des TextTransformers selbst ist die Verwendung von C++ - Kommentaren erlaubt:

In die Variablendeklaration:

```
int iCount, iEnd;
```

können Kommentare eingeschlossen werden, ohne dass der Code damit syntaktisch ungültig würde:

```
int iCount  /* int Variable als Zähler */, iEnd /* Maximalwert */;
```

C++ -Kommentare sind **Textabschnitte, die** zwischen den beiden Token "/*" und "*/" **eingeschlossen sind**. Innerhalb eines solchen Einschlusses kann im Falle von C++ - Kommentaren beliebiger Text stehen, also darf auch das Schlüsselwort "int" darin vorkommen. Es wird aber hier nicht als Variablentyp interpretiert.
In älteren Versionen des TextTransformers wurden Kommentare ausschließlich als Teil der auszulassenden Zeichen behandelt und durch komplexe reguläre Ausdrücke als ganze erkannt. Es ist aber auch möglich, dass die Texte eines Einschlusses nicht beliebig sind, sondern dass sie einer eigenen Grammatik gehorchen. Z.B. gibt es Konventionen, die in C++ - Kommentare Anweisungen einfügt, die als Dokumentation des Programmcodes extrahiert werden können.

Dazu bedarf es dann neben dem Parser für den eigentlichen Programmcode eines zweiten Parsers für die Dokumentation.

Im TextTransformer ist es möglich die Produktionen für diesen zweiten Parser in ein Projekt mit einzufügen und unmittelbar im Wechsel mit dem Hauptparser ausführen zu lassen. Es ist sogar möglich beliebig viele verschiedene Parser ineinander zu schachteln. Diese Parser können jeweils auf eigenen Token-Mengen operieren, so dass z.B. das Token "int" nur vom C++ - Code-Parser erkannt wird, nicht aber vom Kommentar-Parser.

6.11 Unterparser

Eine Produktion kann direkt aus dem Interpreter-Code aufgerufen werden. Sie gehört dann nicht zur eigentlichen Grammatik des Parsers, in den dieser Interpreter-Code eingebettet ist. Die aufgerufene Produktion ist vielmehr eine Startregel für einen gesonderten Parser und diesem wird ein neuer Eingabetext explizit übergeben.

6.12 Familienkonzept

Bei anderen Programmen, den sogenannte Parsergeneratoren, die dem TextTransformer verwandt sind, ist es stets erforderlich, dass es eine Produktion gibt, die allen anderen übergeordnet ist und mit der als Startregel, das Parsen eines Textes stets beginnt.
Demgegenüber ist die Gesamtheit der Produktionen und Token eines TETRA-Projekts ist offen. Aus der Menge der vorhandenen Produktionen kann jede als Startregel ausgewählt werden. Die Regeln, von denen die aktuelle Startregel abhängt werden dann von TETRA automatisch ermittelt und zu einem TETRA-Programm verknüpft.

Beispiel

Ein Projekt kann eine Sammlung von Regeln zur Übersetzung einer Programmiersprache in eine andere enthalten. Im Idealfall wäre eine solche Sammlung vollständig. Dann gäbe es eine derart allgemeine Startregel, dass mit ihr beginnend jedes Programm der Ausgangssprache in die Zielsprache übersetzbar wäre.
Sämtliche für einen vollständigen Übersetzer erforderlichen Regeln zu erstellen ist ein anspruchsvolles Unterfangen und häufig nicht wirklich nötig. Für die gelegentliche Übersetzung von Programmteilen ist es ökonomischer nur die Übersetzung bestimmter Konstrukte zu automatisieren und andere Teile manuell zu transformieren.
Der TextTransformer ist ein gutes Werkzeug, für einen solchen Pool von Regeln.

6.13 Tests

Im TextTransformer können für die Produktionen Testskripte erstellt werden, die dazu dienen das korrekte Funktionieren einzelner Regeln zu überprüfen. Tests können hilfreich sein bei der Erstellung neuer Regeln, sind aber vor allem dazu gedacht sicherzustellen, dass die Verbesserung des Projektes an einer Stelle nicht unvorhergesehenen Auswirkungen an anderer Stelle mit sich bringt.

Ein Test besteht in der isolierten Ausführung einzelner Regeln eines Projektes. Ebenso wie bei der Ausführung des gesamten Projekts wird der zu testenden Regel dabei ein Eingabetext übergeben, der zu einem Ausgabetext transformiert wird. Das Ergebnis der Transformation wird mit dem erwarteten Resultat verglichen. Falls die beiden nicht übereinstimmen wird eine entsprechende Fehlermeldung erzeugt.

TextTransformer

Teil VII

7 Beispiele

Die Beispiele sind wie ein Tutorium aufgebaut und eignen sich zum ersten Kennenlernen des TextTransformers.

Wortvertauschung
Konvertierung eines Atari-Textes
Rechner
Textstatistik
GrepUrls
E-Mail-Adresse
Guard
Rechnung
XML
Java

Zu den Beispielen gehören Parser mit denen der TextTransformer selbst arbeitet. Abgesehen von *EditProds* sind die Parser vom semantische Code befreit.

TETRA-Produktionen
TETRA-EditProds
TETRA-Interpreter
TETRA-Import

Cocor Import

Weitere Beispiele gibt es unter:

http://www.text-konverter.homepage.t online.de

Videos gibt es bisher nur mit englischen Sprechblasen unter

http://www.texttransformer.com/Videos_en.html

7.1 Wortvertauschung

Anhand des sehr einfachen Beispiels der Vertauschung zweier Worte im Text werden die wesentlichen Teile und Arbeitsschritte, die für ein TETRA-Programm notwendig sind kurz dargestellt.

Problemstellung:

Bei der Erstellung eines Textes kann es immer wieder vorkommen, dass unbeabsichtigt zwei Ausdrücke verwechselt werden, seien es die Namen zweier Personen: Marcuse und Mabuse, seien es zwei Fremdworte: ontologisch und ontisch, oder die Namen zweier chemischer Verbindungen N,N-Di-(2-hydroxyethyl)-N',N'-dimethyl-3,7-diaminophenothiazoniumjodid und N,N'-Di-(2-hydroxyethyl)-N,N'-dimethyl-3,7-diaminophenothiazoniumjodid. Diese Verwechslung soll korrigiert werden.

Herkömmliche Korrekturmethode:

Wollte man diesen Fehler mit der Methode des Suchens und Ersetzens in einer Textverarbeitung korrigieren, so müsste zunächst einer der beiden Ausdrücke durch einen dritten ersetzt werden (z.B. Mabuse durch Labuse), dann der andere Ausdruck durch den ersetzten (Marcuse durch Mabuse) und schließlich der dritte (Labuse) durch den zweiten (Marcuse).

TETRA Programm:

In TETRA ist die Vertauschung zweier Begriffe in einem Arbeitsgang möglich. Allerdings muss hierfür ein kleines Programm - bestehend aus nur einer Regel - geschrieben werden. Der Aufwand macht sich dann schnell bezahlt, wenn man mehrere Texte mit vertauschten Begriffen zu korrigieren hat. Zudem ist es möglich in demselben Durchgang zugleich mehrere vertauschte Wortpaare zu korrigieren.
Dass TETRA mit einem Arbeitsgang auskommt, liegt daran, dass der Text hier strikt von links nach rechts durchlaufen wird und immer dann, wenn einer der vertauschten Begriffe angetroffen wird, direkt die Ersetzung durch den jeweils anderen erfolgt.

Es werden **zwei Versionen** für das Wortvertauschungsprojekt vorgestellt:
a) eine Version, die mit nur einer Produktion auskommt
b) eine Version, bei der die zentrale Produktion vereinfacht wird und zusätzlich Token auf der Tokenseite definiert werden.

7.1.1 Öffnen und Ausführen eines Projekts

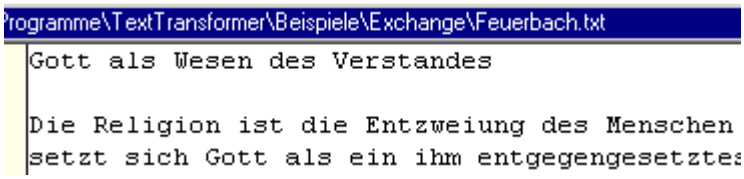
Zum Testen des Projekts befindet sich ein Textabschnitt des Philosophen Ludwig Feuerbach in dem Unterordner *Exchange* des Beispielverzeichnisses von TETRA:

"\TextTransformer\Beispiele\Exchange\Feuerbach.txt"

In diesem Text sollen die Worte "Mensch" und "Gott" sowie "Menschen" und "Gottes" vertauscht werden.

Öffnen sie zunächst den Text über das Menü mit *Datei->Öffnen*. Zur übersichtlicheren Darstellung des Textes empfiehlt es sich den Zeilenumbruchsschalter

zu betätigen. Die langen Textzeilen werden umgebrochen und der gesamte Text wird lesbar.



Das Projekt, das das Beispielsprogramm zur Wortvertauschung enthält befindet sich im gleichen Ordner:

"\TextTransformer\Beispiele\Exchange\Exchange.ttp"

Öffnen sie das Projekt über das Menü mit *Datei->Projekt öffnen*. Im rechten Fensterblock erscheint im Register *Syntaxbaum* der Name der Regel (Produktion): *Exchange*.

Bevor erklärt werden soll, was eine Produktion ist, können sie das Programm schon einmal ausführen, um zu sehen wie Texte in der TextTransformer IDE transformiert werden können.

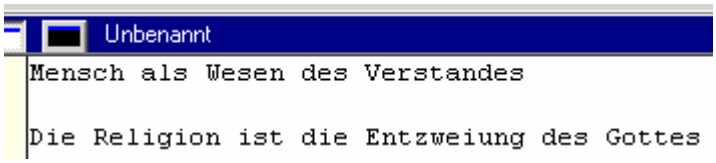
Betätigen sie in der Werkzeugleiste den Schalter zum Ausführen des Programms:

Ein Fortschrittsbalken zeigt den Verlauf der Transformation an. Er wird schnell wieder verschwinden, da der Quelltext sehr kurz ist, und das Ergebnis der Textumwandlung steht im Ausgabefenster.

Wenn sie mit der Maus auf die Trennungslinie von Ein- und Ausgabefenster gehen, wechselt der Mauscursor seine Form
:

Bei gedrückter Maustaste kann das untere Fenster nun auf die gleich Größe gebracht werden wie das Eingabefenster. Klicken sie in das untere Ausgabefenster und aktivieren sie wie zuvor beim Eingabefenster den Zeilenumbruch.

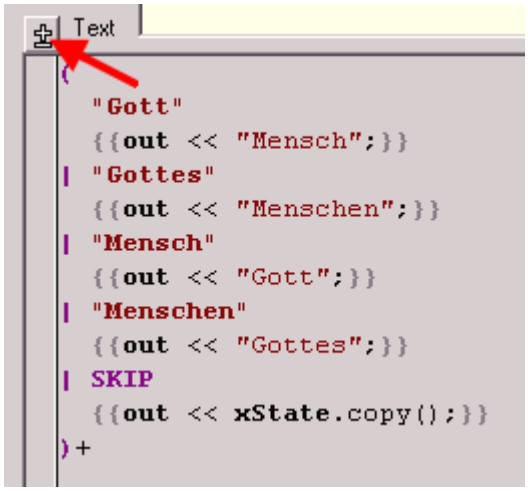
Im Menü unter *Einstellungen->Fenster synchronisieren* können sie dafür sorgen, dass im Ein- und Ausgabefenster beim Scrollen jeweils die gleichen Zeilenbereiche angezeigt werden.



Der Vergleich des Ein- und Ausgabetextes zeigt, dass die materialistische Philosophie Feuerbachs in eine Art verkehrten Idealismus transformiert wurde und dies durch bloßen Austausch der Worte: "Gott" und "Mensch" und der zugehörigen Genitiva.

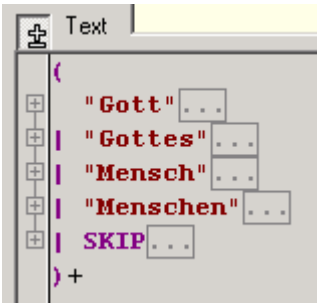
7.1.2 Produktion

Wenn sie im rechten Fensterblock direkt auf den Namen der Regel *Exchange*. klicken, wird im linken Fensterblock automatisch in das Registerfenster *Produktionen* gewechselt und die Eigenschaften der *Exchange*-Regel werden angezeigt. Hier interessiert besonders der Text der Produktion:



Die Syntax und Bedeutung der TETRA-Produktionen wird in der Hilfe im Kapitel Skripte ausführlich dargestellt. In den folgenden Erklärungen soll nur ein erster Eindruck der Funktionsweise eines Regelskripts geboten werden.

Zunächst soll nur der Teil des Codes besprochen werden, der die Analyse des Textes vornimmt und von dem Teil abgesehen werden, der für die Umformung des Textes steht.
Dazu klicken Sie bitte auf den mit dem roten Pfeil markierten Schalter zum Einklappen des semantischen Code. Nun erhalten Sie folgendes Bild:



(...)+

Die gesamte Regel ist in die Klammern: (...)+ eingeschlossen. Dies bedeutet, dass der durch die Regel beschriebene Quelltext aus einer ein- oder mehrmaligen Wiederholung dessen besteht, was innerhalb der Klammern beschrieben wird.

'|'

Innerhalb der Klammern befinden sich fünf Abschnitte, die durch das sogenannte Pipe Zeichen '|' voneinander getrennt sind. Dieses Zeichen trennt Alternativen. Der gesamte Text besteht also aus der ein- oder mehrmaligen Wiederholung jeweils einer von fünf Alternativen. Sieht man zunächst von den Ausdrücken in den doppelten geschweiften Klammern ab, so sind diese Alternativen:

"Gott" | "Gottes" | "Mensch" | "Menschen" | SKIP

D.h. an einer Textstelle steht entweder das Wort "Gott" oder "Gottes" oder "Mensch" oder "Menschen" oder die fünfte Alternative, die mit "SKIP" bezeichnet ist, trifft zu.

SKIP

Mit dem Schlüsselwort SKIP wird TETRA angewiesen all denjenigen Text bei der Analyse zu überspringen, der nicht eine der Alternativen zu 'SKIP' darstellt; also im gegenwärtigen Beispiel sämtlicher Text, der nicht in den Worten "Gott", "Gottes", "Mensch" oder "Menschen" besteht. Nun ist es logisch, dass die Regel "Exchange" den gesamten Eingabetext abdecken muss. Der gesamte Text besteht aus einem der vier Worte oder eben aus anderen Worten.

{{..}}

Es bleiben nur noch die Regelteile zu erklären, die in den doppelt geschweiften Klammern der ersten Abbildung stehen. Sie sind die Anweisungen die ausgeführt werden, sobald die jeweils voranstehende Alternative erkannt wurde.

"out <<"

Die Anweisung "out <<" bedeutet, dass der jeweils nachfolgende Ausdruck in die Ausgabe geschrieben werden soll. Wurde also das Wort "Gott" erkannt, so wird "Mensch" in die Ausgabe geschrieben, wurde umgekehrt das Wort "Mensch" erkannt "Gott".

xState.copy()

Im Falle der Skip-Alternative wird xState.copy() in die Ausgabe geschrieben. *xState* repräsentiert

den jeweiligen Zustand des Transformationsprozesses und xState.copy() gibt den zuletzt erkannten Textabschnitt zurück.

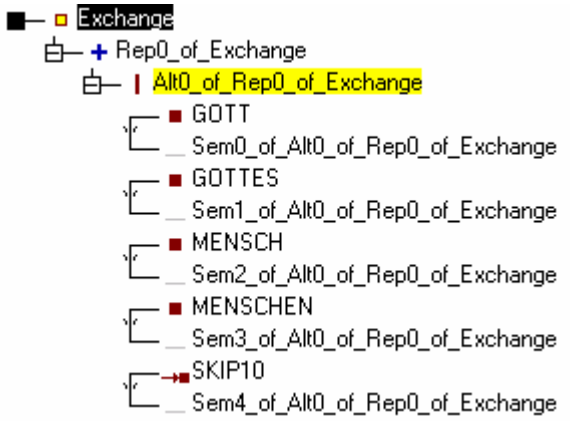
7.1.3 Schrittweise Analyse

Die Erklärung der Regel lässt sich im einzelnen demonstrieren, indem das Programm schrittweise ausgeführt wird.

Klicken sie dazu auf den *Tetra*-Reiter des linken Fensterblocks.

Nun kann durch wiederholte Betätigung des Schalters *Einzelne Anweisung*

das Programm schrittweise ausgeführt werden. Nach der ersten Betätigung öffnet sich eine Baumansicht unter dem Namen der *Exchange*-Produktion, die die Syntax dieser Regel abbildet. Nach einer zweiten Betätigung des gleichen Schalters öffnet sich der Baum noch weiter, so dass folgendes Bild erscheint:



Der Knoten *Exchange* steht für die Regel selbst.
Der Knoten *Rep0_of_Exchange* steht für die Klammer "(...)+"
Der Knoten *Alt0_of_Rep0_of_Exchange* steht für die Menge der Alternativen.
Jede der Alternativen besteht aus zwei Knoten: einem Knoten für den zu erkennenden Text und einen weiteren Knoten für die auf die Erkennung folgende semantische Aktion.

Bei jedem weiteren Schritt wandert die gelbe Markierung auf den jeweils nächsten Baumknoten, der zur Erkennung des nächsten Abschnitts des Eingabetextes führt. Immer, wenn ein Knoten verlassen wird, der eine semantische Aktion repräsentiert, wird die Anweisung ausgeführt, d.h. hier, dass der Ausgabetext entsprechend ergänzt wird.

Nach der schrittweisen Ausführung eines Programms, die im sogenannte Debug-Modus erfolgt,

muss der TextTransformer wieder in den Normal-Modus zurückgesetzt werden. Dies geschieht mit dem Schalter:

Dabei wird auch der Ausgabetext gelöscht.

7.1.4 Token verwenden

Das Exchange-Projekt enthält noch eine zweite Produktion: ***Exchange_Token***.
Exchange_Token ist eine alternative Möglichkeit die Wortvertauschungen zu programmieren.
In dieser Version werden die auszutauschenden Begriffe auf der Token-Seite definiert, um sie direkt mit Aktionen zu verbinden.

Für das Wort "Gott" wird folgendes Token definiert:

Die Aktion zur Ausgabe von "Mensch" wird bei einer Transformation stets dann automatisch ausgeführt, wenn das Textmuster "Gott" erkannt wurde. Nach analogen Tokendefinitionen für die übrigen Begriffe vereinfacht sich die ursprüngliche Definition der Produktion:

```
Exchange_Token =
  (
    GOTT
  | GOTTES
  | MENSCH
  | MENSCHEN
  | SKIP
    {{out << xState.str();}}
  )+
```

Die Ausführung dieser Produktion erfolgt wieder auf der *Tetra*-Seite. Zunächst muss *Exchange_Token* als Startregel gewählt werden:

Dann wird wie oben der Schalter *Einzelne Anweisung* betätigt.

Die Darstellung der Produktion im Syntaxbaum hat sich jetzt vereinfacht, da die semantischen Aktionen nicht mehr angezeigt werden:

Bei einem so einfachen Projekt wie diesem ist es Geschmackssache, ob man es in der Art der *Exchange*- oder der *Exchange_Token*-Produktion formuliert. Soll jedoch in einem komplexeren Projekt nach der Erkennung eines Tokens, das eventuell in verschiedenen Produktionen vorkommen kann, stets die gleiche Aktion ausgeführt werden, ist die Verwendung der mit dem Token verknüpften Aktion angebracht.

7.2 Konvertierung eines Atari-Textes

Das Beispiel *Konvertierung eines Atari-Textes* ähnelt dem vorhergehenden. Hier werden jedoch nicht Worte, sondern spezielle Zeichen ausgetauscht.

Problemstellung:

Texte, die auf einem Atari-Computer geschrieben wurden, sollen in einem Texteditor unter Windows weiterverarbeitet werden.

TETRA Programm:

Sie finden einen solchen Text: *Test.txt*, in dem Unterordner *Atari* des Beispielverzeichnisses von TETRA:

\TextTransformer\Beispiele\Atari1\Test.txt

Öffnen Sie ihn über das Menü mit *Datei->Öffnen*.

Nachdem der Textes geöffnet wurde, sieht sein Anfang wie folgt aus:

```
7
8           Nachfahrenliste des Kaufmannes
9           ------------------------------
10      Barthold Jacob □Benjamin□ MEYER aus Hamburg
11      ------------------------------------------
12
13  B.J.B. MEYER hat (vermutlich aufgrund von Tagebuchnotizen) im
14  Alter eine Chronik □ber sein Leben geschrieben. Sein Enkel,
```

Der Text enthält Zeichen, die nur durch ein leeres Quadrat "□" dargestellt sind. Dies sind zum einen Zeichen, die in der Atari-Textverarbeitung Textattribute (fett, kursiv, unterstrichen) definieren. Zum anderen sind es Sonderzeichen wie die Umlaute, die nicht richtig dargestellt werden. Der Text muss also konvertiert werden.

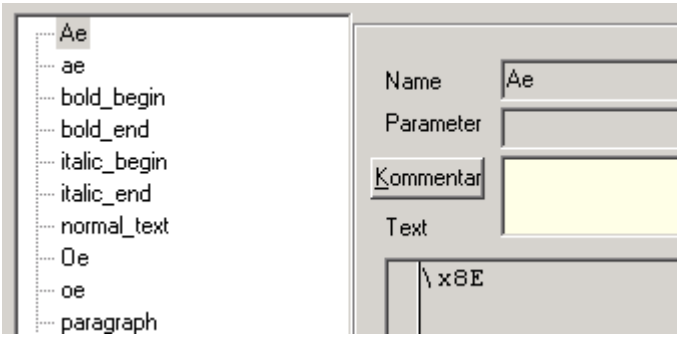
Das TETRA-Projekt, mit dem sich eine solche Konvertierung durchführen lässt, können Sie über das TETRA-Menü mit "Datei->Projekt öffnen" laden. Es befindet sich im gleichen Verzeichnis wie der Test-Text:

"\TextTransformer\Beispiele\Atari\Atari.ttp".

Es werden nacheinander die Definitionen der Token, der Produktionen und der Aktionen vorgestellt. Schließlich wird noch eine zweite Variante des Projekts präsentiert, die den Eingabetext in das RTF-Format transformiert.

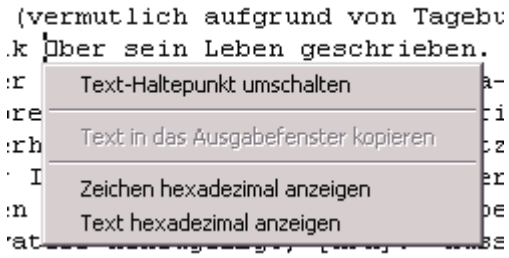
7.2.1 Token

Um die nicht lesbaren Zeichen in lesbare zu übersetzen, müssen sie zunächst im Text gefunden werden. Die Textelemente bzw. Token, die das Übersetzungsprogramm zu suchen hat werden auf der zweiten Seite des TETRA-Programms definiert. Wenn sie mit der Maus den *Token*-Reiter des Registers anklicken, erscheint am linken Rand der Seite eine Liste von Namen der definierten Terminalsymbole.

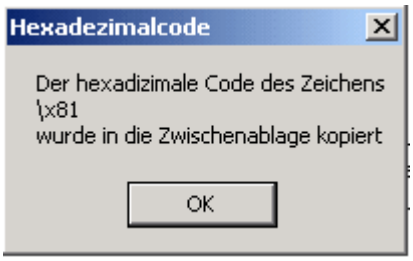


Wird ein Name der Liste angeklickt, so erscheint im Textfenster die Definition des entsprechenden Symbols. Mit einer Ausnahme sind sämtliche Definitionen des Atari-Projekts einander ähnlich: ein Backslash '\' gefolgt von einem 'x' und zwei Ziffern. Wird z.B. das Symbol **ue** angeklickt, so erscheint im Textfenster der Ausdruck: **\x81**. Dieser Ausdruck ist eine Nummer in hexadezimaler Schreibweise, die dem Zeichen im ANSI-Zeichensatz zugeordnet ist. Statt dieses Ausdrucks könnte auch das leere Quadrat "□", das an einer Stelle des Textes vorkommt, die durch ein 'ü' ersetzt werden soll im Textfenster stehen. Dann sähen die Textfenster für alle Umlaute jedoch gleich aus.

Der Ausdruck: "\x81", lässt sich leicht ermitteln, indem zur *TETRA*-Seite zurückgegangen wird und nach Platzierung des Textcursors links neben dem unbekannten Zeichen die rechte Maustaste gedrückt wird. Hier erscheint dann ein Popup-Menü in dem der Punkt ***Zeichen hexadezimal anzeigen*** betätigt wird.



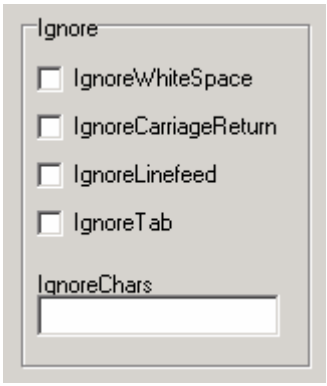
Nun erscheint eine Dialogbox mit dem gesuchten Ausdruck. Zugleich wird der Ausdruck in die Zwischenablage kopiert, von wo aus es in den Text zur Definition eines Symbols eingefügt werden kann.



Eine Symboldefinitionen unterscheidet sich von den übrigen:

normal_text =
[^\x11\x12\x15\x16\x17\x18\x81\x84\x94\x99\x9E\x8E\x9A\x9C]+

Dieser Ausdruck definiert die Textabschnitte, die kein Sonderzeichen und kein Textattribut sind. Im inneren der eckigen Klammer befindet sich ein Negationszeichen '^' gefolgt von der Liste der einzelnen in den übrigen Symboldefinitionen vorkommenden Hexadezimalausdrücken. Die eckigen Klammern definieren eine Zeichenmenge; hier die Menge aller Zeichen, die kein Sonderzeichen und kein Textattribut sind, also z.B einen Buchstaben des Alphabets oder ein Satzzeichen. Das der eckigen Klammer folgende Pluszeichen bedeutet, dass Zeichen der vorangehenden Menge einmal oder beliebig oft aufeinander folgend auftreten.
normal_text schließt auch Zeilenumbruchszeichen, Tabulatoren und Leerzeichen mit ein. Standardmäßig werden diese Zeichen in TETRA-Projekten ignoriert. Für das Atari-Projekt wurde diese Standardeinstellung geändert. Im Menü unter "Einstellungen->Projektoptionen" sind daher die Markierungen für alle ignorierbaren Zeichen gelöscht.



7.2.2 Produktionen

Während das Wortvertauschungs-Beispiel mit nur einer Produktion auskam, besteht das Atari-Projekt aus drei Produktionen. Es wäre möglich gewesen auch hier mit nur einer Produktion auszukommen, sie wäre dann allerdings recht umfangreich.
Bei den nachfolgenden Erklärungen wird zunächst von den Texten, die von den Klammern "{=" und "=}" begrenzt werden abgesehen (siehe Abschnitt "Aktionen").

Die Produktion **sonderzeichen** besteht aus den alternativen Symbolen der Sonderzeichen.

```
paragraph  | ae  | Ae | oe  | Oe  | ue   | UE   | sz
```

Die Alternativbeziehung wird durch das Zeichen '|' ausgedrückt.
Die Produktion *sonderzeichen* steht an den Positionen in Übereinstimmung mit dem zu konvertierenden Text, an denen eines der Sonderzeichen steht.

Die Regel **textattribut** besteht aus den alternativen Symbolen für die Zeichen durch die die Atari-Textattribute definiert werden.

```
  bold_begin
| bold_end
| italic_begin
| italic_end
| underline_begin
| underline_end
```

Die Regel *textattribut* steht an den Positionen in Übereinstimmung mit dem zu konvertierenden Text, an denen eines der Zeichen steht, durch die die Textattributsymbole definiert sind.

Schließlich die Startproduktion: **Atari**.

```
(
  sonderzeichen
| textattribut
| normal_text
)+
```

Wer das erste Beispiel aufmerksam studiert hat, dem wird vielleicht die strukturelle Ähnlichkeit zwischen dem **normal_text**-Token dieses Beispiels und dem **SKIP**-Knoten des Exchange-Beispiels auffallen. Tatsächlich wurde das *normal_text*-Token hier nur zu didaktischen Zwecken definiert und die Produktion hätte auch analog zum Wortvertauschungs-Beispiel formuliert werden können:

```
(
  sonderzeichen
| textattribut
| SKIP
)+
```

Das **normal_text**-Token soll allen Text umfassen, der kein Textattribut und kein Sonderzeichen darstellt und der **SKIP**-Knoten des vorigen Beispiels sollte allen Text umfassen, der nicht aus den Worten "Gott", "Gottes", "Mensch" oder "Menschen" besteht.

7.2.3 Aktionen

Wie im ersten Beispiel, so wird auch hier nach jedem erkannten Textabschnitt (Zeichen) eine semantische Aktion ausgeführt, in der ein Text in die Ausgabe geschrieben wird.

Der vom *normal_text*- (SKIP-) Token erkannte Text wird kopiert:

```
{{out << xState.copy(); }}
```

Der von einem Sonderzeichen erkannte Text wird in ein lesbares Zeichen übersetzt, z.B.:

```
| ue   {= out << "ü"; =}
```

Bei den Textattributen allerdings sind keine semantischen Aktionen angeführt. Das bewirkt ein Überspringen dieser Attribute. Jedes Textverarbeitungsprogramm hat seine eigene Art diese Attribute in seinen Textdokumenten zu kodieren. Als Beispiel wird unten eine Konvertierung in das RTF-Format besprochen.

Das was mit den jeweils erkannten Textabschnitten geschehen soll, steht in den Regelskripten innerhalb der Klammerpaare "{=" und =}" die in der bisherigen Erläuterung übersprungenen wurden. Innerhalb der Klammern werden sogenannte "semantische Aktionen" definiert. In der Baumansicht werden sie durch die Knoten dargestellt, deren Namen mit "Sem" und angehängten Ziffern bezeichnet sind; z.B. *Sem0_of_Alt0_of_Rep0_of_Atari* .

Die Befehle für die semantischen Aktionen stellen eine Untermenge der Befehle der Programmiersprache C++ dar. Im *Atari*-Projekt wird nur ein Befehl verwendet: das "Schieben" eines Textes in die Ausgabe.

Beispielsweise enthält die Produktion **sonderzeichen** die Zeile:

```
  | ue   {= out << "ü"; =}
```

Das bedeutet: sobald das Symbol *ue* erkannt ist führe die Aktion

```
  out << "ü";
```

aus, d.h. hänge den Text "ü" an den Ausgabetext an. Ähnlich lauten die semantischen Aktionen die nach Erkennung der anderen Symbole ausgeführt werden sollen.
Bei den Textattributen allerdings sind keine semantischen Aktionen angeführt. Das bewirkt ein Überspringen dieser Attribute. Jedes Textverarbeitungsprogramm hat seine eigene Art diese Attribute in seinen Textdokumenten zu kodieren. Als Beispiel wird unten eine Konvertierung in das RTF-Format besprochen.

Wenn sie nun das Programm ausführen, sieht der eingangs gezeigte Textausschnitt so aus:

```
8          Nachfahrenliste des Kaufmannes
9          ------------------------------
10     Barthold Jacob Benjamin MEYER aus Hamburg
11     ----------------------------------------
12
13 B.J.B. MEYER hat (vermutlich aufgrund von Tagebuchnotizen) im
14 Alter eine Chronik über sein Leben geschrieben. Sein Enkel,
```

Die Textattributzeichen sind entfernt und ein "ü" hat den Platz eines zuvor unlesbaren Zeichens eingenommen.

7.2.4 Konvertierung nach RTF

Bis jetzt wurden die Zeichen, die die Atari Textattribute ausmachen: Unterstreichen, Fettdruck und Kursivschrift, ignoriert, da einfache Textdateien solche Attribute nicht enthalten können. Ein Beispiel eines Dokuments, das derartige Attribute enthalten kann, ist eine RTF-Datei, die nach dem Rich Text Format: RTF, aufgebaut ist.
Das Projekt

...\Tetra\Beispiele\Atari\Atari2Rtf.ttp

zeigt, wie Atari Texte in das Rich Text Format transformeiert werden können. Die relevanten Teile RTF-Spezifikation werden unmittelbar verwendet, ohne diese Spezifikation im einzelnen zu erklären. Eine gute Einführung ist "RTF Pocket Guide" von Sean M. Burke.

http://www.oreilly.com/catalog/rtfpg/

Dort kann das wichtige erste Kapitel des Buchs als pdf-Datei heruntergeladen werden. Die originale Rich-Text-Spezifikation ist zu finden unter:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnrtfspec/html/rtfspec.asp

Gemäß der RTF Spezifikation werden Textteile mit besonderen Attributen in die Klammern "{" und "}" eingeschlossen und an den Anfang des eingeschlossenen Textes wird ein Ausdruck für das Attribut geschrieben:

kursiv: \i
fett: \b
unterstrichen: \ul

So werden die Token-Aktionen zu:

```
italic_begin: out << "{\\i ";
italic_end: out << "} ";
```

Im RT-Format werden Sonderzeichen durch ein Backslash gefolgt von einem Apostroph und dem Hexadezimal-Code des Zeichens ausgedrückt: So wird die Token-Aktion z.B. von *ae* zu:

```
ae: out << "\\'e4";
```

Zeilenumbrüche, die im angezeigten Text sichtbar sein sollen, können nicht direkt in eine RTF-Datei geschrieben werden.. Sie müssen durch "\line" oder "\par" kodiert wrden. Deshalb muss ein zusätzliches Token definiert werden, das die Zeilenumbrüche erkennt:

Name: EOL
Text: \r?\n
Aktion: out << endl << "\\line ";

(Durch *endl* wird ein Zeilenumbruch in die RTF-Quelle geschrieben, um seine logische Struktur besser sichtbar zu machen. Als Zeilenumbruch angezeigt wird nur "\line")

Das gesamte RTF-Dokument muss in die Klammern "{" und "}" eingeschlossen werden und auf die

erste Klammer folgt ein Header, der spezifiziert, um welche RTF-Version es sich handelt, welche Fonts verwendet werden und welche Schriftgröße dargestellt werden soll. Der Anfang eines RTF-Dokuments kann dann z.B. so aussehen:

```
{\rtf1\ansi\deff0 {\fonttbl{\f0 Courier New;}}\f0\fs20
```

Um diesen Header zu schreiben soll erstmals die Seite für die Klassenelemente des TextTransformers benutzt werden. Diese Seite dient dazu Variablen und Funktionen zu definieren, die dann überall im Projekt verwendet werden können. Für das Atari-Projekt werden drei Funktionen definiert:

RtfBegin:

```
out << "{\\rtf1"    // RTF, version 1
    << "\\ansi"     // ANSI character set
    << "\\deff0 ";  // default font #0
```

RtfFont:

```
out  << "{\\fonttbl" // font table
     << "{\\f0 Courier New;}" // font #0
     << "}"; // font table end
out  << "\\f0"    // use font #0
     << "\\fs20";  // font size = 20/2 = 10 points
```

RtfEnd:

```
out << "}";
```

RtfBegin und *RtfFont* hätten auch zu einer einzelnen Funktion kombiniert werden können oder auch direkt in die *Atari*-Produktion geschrieben werden können. Die Aufteilung in mehrere Funktionen macht die logische Struktur des Projekts aber übersichtlicher.

Die Atari Produktion ist nun:

```
{{
  RtfBegin();
  RtfFont();
}}
(
    sonderzeichen
  | textattribut
  | SKIP {= out << xState.str(); =}
)+
{{
  RtfEnd();
}}
```

Nachdem ein Text transformiert wurde, muss er als RTF-Datei gespeichert werden, d.h. der Dateierweiterung muss "rtf" sein. Nun kann die Datei mit einem Textverarbeitungsprogramm geöffnet werden - z.B. Wordpad -, das in der Lage ist, RTF-Dokumente darzustellen. Dort sieht man nun z.B. *Benjamin* korrekt in Kursiv-Schrift geschrieben:

Jacob *Benjamin* MEYER

7.3 Rechner

Die wichtigsten Bedienungselemente von TETRA sollten bekannt sein.

Problemstellung:

Aus Rechenausdrücken wie "(3.2 + 8.9 - 4.6) * 5.6" sollen die Ergebnisse berechnet werden.

TETRA Programm:

Bei dem Rechner-Beispiel handelt es sich um eine klassische Anwendung eines Parsers. Aus dem Eingabetext müssen die Zahlen, Operatoren und Klammerungen extrahiert werden, um das Ergebnis berechnen zu können. Das Projekt besteht aus einigen Token und aus insgesamt fünf Regeln die im folgenden einzeln erläutert werden.

Auch für dieses Projekt gibt es wieder **zwei Versionen**:

a) eine Version, in der die Verwendung von Parametern demonstriert wird

...\TextTransformer\Beispiele\Rechner\Rechner1.tpp

b) eine zweite die die Möglichkeit zur Rückgabe von Werten ausnutzt

...\TextTransformer\Beispiele\Rechner\Rechner2.tpp

7.3.1 Token

Die Token für die Operatoren und Klammern werden direkt innerhalb der Regeln definiert:

```
"+", "-", "*","/","(", ")"
```

Ein komplexes Token zur Erkennung der Zahlen wird auf der Tokenseite als regulärer Ausdruck definiert:

```
number  = \d+(\.\d*)?|\.\d+
```

Hierin bedeutet:

"\d" eine einzelne Ziffer
"\d+" eine beliebig lange Folge von Ziffern bestehend aus mindestens einer Ziffer
"\." der Punkt (ohne den vorangestellten Backslash '\' hätte der Punkt eine Sonderbedeutung innerhalb des regulären Ausdrucks.
"\d*" eine beliebig lange Folge von Ziffern, die auch leer sein kann
"?" ein optionales Vorkommen des davor stehenden Klammerinhalts
"|" eine Alternative

Fasst man dies alles zusammen, so ist eine Zahl (number) entweder eine Folge von Ziffern optional gefolgt von einem Punkt gefolgt von null oder mehr weiteren Ziffern (z.B.:: "123" oder "123." oder "123.45"). Oder eine Zahl ist ein bloßer Punkt gefolgt von eine Ziffernreihe (z.B..: ".45")

7.3.2 Produktion: Rechner

Die Startregel für das Rechner-Programm ist:

```
{{double d;}}
Expression[d]
{{out << d << "\n";}}
```

Die Aktionen werden in diesem Beispiel durch doppelte geschweifte Klammern **{{...}}** eingeschlossen. Für so geklammerte Aktionen ist in den Projekteinstellungen (über das Menü erreichbar) einstellbar, ob sie innerhalb der TETRA-Entwicklungsumgebung unmittelbar ausführbar, d.h. interpretierbar sein sollen oder nicht. Die Standardeinstellung, dass sie interpretierbar sein sollen wird im Rechner-Projekt beibehalten.

Die Aktion:

```
double d;
```

stellt eine Variable des Typs double bereit (deklariert sie), die das Ergebnis der Rechnung aufnehmen soll. Double-Variablen sind dadurch ausgezeichnet, dass sie Zahlen mit Vor- und Nachkommastellen enthalten können.

Die Aktion:

```
out << d << "\n";
```

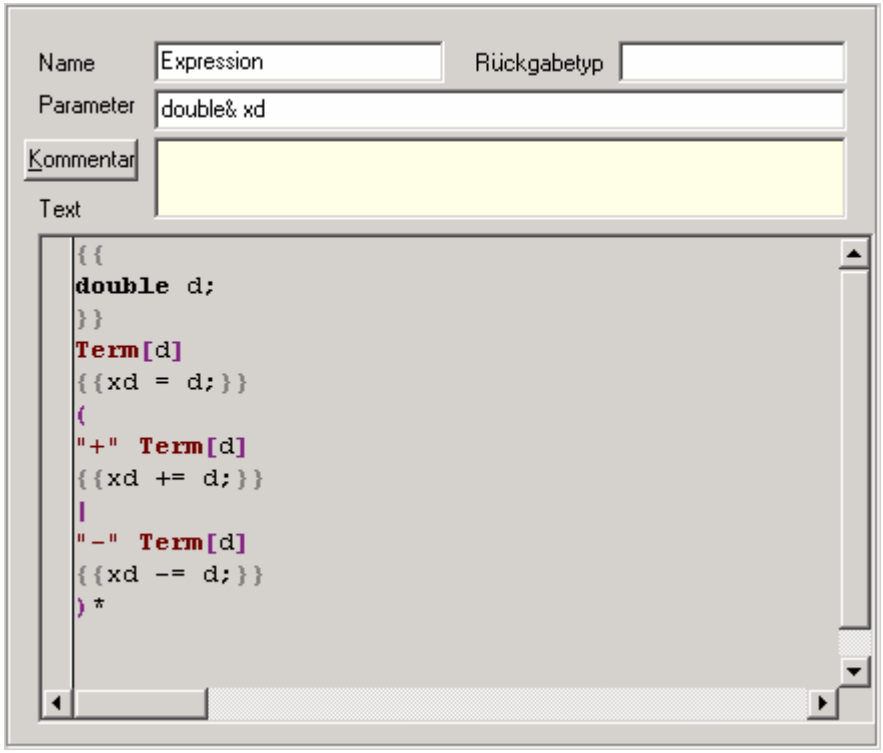
ist schon aus den vorherigen Beispielen bekannt. Der Wert der Variablen wird gefolgt von einem Zeilenumbruch in die Ausgabe geschrieben.

Zwischen den beiden Aktionen wird die Regel *Expression* ausgeführt, innerhalb derer die Berechnung des Ergebnisses erfolgt. Damit das Ergebnis in die Variable "d" geschrieben werden kann, muss sie an die Expression-Regel übergeben werden. Dies ist die Bedeutung der **eckigen Klammern** hinter dem Regelnamen.

```
Expression[d]
```

7.3.3 Produktion: Expression

Die *Expression*-Regel hat einen Parameter:



Parameter: `double& xd`

Dies ist die Schnittstelle an der die in der Rechner-Regel definierte Variable in die Expression-Regel hineingereicht wird. Innerhalb der neuen Regel hat sie den neuen Namen xd. Das Wort "double" bedeutet wiederum den Typ der Variablen. Das "&" kennzeichnet den Parameter als eine Referenz-Variable, d.h. dass die Variable auch wieder herausgereicht wird. Nach Beendigung der Expression-Regel kann der Inhalt der Variablen innerhalb der Regel, die die Expression-Regel aufgerufen hat weiterverarbeitet werden. Ohne das "&" würde die übergebene Variable innerhalb der aufrufenden Regel ihren Wert beibehalten, auch wenn ihr Wert, der an die Unterregel übergeben wurde dort verändert wurde.

Sieht man im Text der Produktion zunächst von den Aktionen ab, so wird die Regel übersichtlicher:

```
Term
(
"+" Term
|
"-" Term
)*
```

Ein Rechenausdruck (Expression) besteht aus einem Term, zu dem eine beliebige Anzahl weiterer Terme hinzuaddiert abgezogen werden.

Die Anweisung:

```
double d;
```

definiert eine neue Variable des Typs double, die den Inhalt eines einzelnen Terms aufnehmen soll.

Das Ergebnis des ersten Terms wird zunächst an die Referenzvariable *xd* übergeben:

```
xd = d;
```

Die Werte der folgenden Terme werden nun zu dem bisherigen Ergebnis hinzuaddiert bzw. abgezogen.

`xd += d;` bzw. `xd -= d;`

Diese Ausdrücke sind eine abkürzende Schreibweise für:

`xd = xd + d;` bzw. `xd = xd - d;`

7.3.4 Produktionen: Term und Factor

Die ***Term***-Produktion ist analog der *Expression*-Produktion aufgebaut, nur dass hier die Multiplikation bzw. Division an Stelle der Addition und Subtraktion stehen:

```
{{double d;}}
Factor[d]
{{xd = d;}}
(
"*" Factor[d]
{{xd *= d;}}
|"/" Factor[d]
{{
if(d == 0)
   throw CTT_Error("Division durch Null");
xd /= d;
}}
)*
```

Falls der Nenner gleich Null ist, wird das Programm durch den Aufruf von:

```
throw CTT_Error("Division durch Null");
```

abgebrochen. Der Text "Division durch Null" wird dann in der Log-Fenster angezeigt.

Die Produktion ***Factor*** lautet:

```
{{
bool bPlus = true;
double d;
}}
```

```
(
"-"
{{bPlus = false;}}
)?
(
Number[d]
|
"(" Expression[d] ")"
)
{{
if(bPlus)
  xd = d;
 else
  xd = -d;
}}
```

Wenn der Schalter

zum Einklappen des semantischen Codes betätigt wird, ist die Struktur der Produktion besser zu erkennen:

```
( "-" )?
( Number | "(" Expression ")"  )
```

Klammerausdrücke denen ein Fragezeichen nachgestellt ist **"(...)?"** bedeuten in der TETRA-Syntax, dass das was innerhalb der Klammern vorkommt optional ist, also im Text vorkommen kann aber nicht muss.
Ein *Faktor* besteht daher aus einem optionalen Minus-Vorzeichen gefolgt von entweder einer Zahl, die in der *Number*-Produktion ermittelt wird, oder einem Klammerausdruck.
Hier wird der Parser **reflexiv**. Die *Expresssion*-Produktion hatte die *Term*-Produktion aufgerufen, und diese die Factor-Produktion. Innerhalb der *Factor*-Produktion kann es erneut zu einem Aufruf der *Expression*-Produktion kommen. Diese Reflexivität spiegelt die mögliche Verschachtelung von Klammern in Rechenausdrücken wieder, z.B. "3 + ((3.2 + 8.9 - 4.6) * 5.6)"

Das Minus-Vorzeichen ist optional. Sein Vorhandensein wird durch eine boolsche Variable festgehalten. Variablen des Typs **bool** können nur einen der beiden Werte true oder false, d.h. wahr oder falsch enthalten. Die Variable **bPlus** wird am Beginn der *Factor*-Produktion deklariert und auf den Wert true gesetzt:

```
bool bPlus = true;
```

Wird nun das Minuszeichen an der aktuellen Position im Text gefunden, so wird der Wert der Variablen in *false* geändert. Nach Auswertung der *Number*-Produktion oder eines Klammerausdrucks, wird dessen Ergebnis je nach dem Wert der Variablen ***bPlus*** belassen wie er ist oder negiert. Hierzu dienen die Anweisungen:

```
if(bPlus)
  xd = d;
 else
  xd = -d;
```

Die Struktur

```
if ( <bedingung> )
  <anweisung1>;
 else
  <anweisung2>;
```

bedeutet, dass wenn die Bedingung "bedingung" erfüllt ist, d.h. den Wert *true* hat, die Anweisung "anweisung1" ausgeführt wird und anderenfalls "anweisung2".

7.3.5 Produktion: Number

Die *Number*-Produktion schließlich besteht nur aus dem Terminalsymbol **number** (= "\d+(\.\d*)?|\.\d+" s.o.) und einer Aktion, die den Text den das Symbol abdeckt in eine Zahl übersetzt:

```
number
{{ xd = stod(xState.str()); }}
```

Zu dieser Übersetzung dient die spezielle Funktion **stod**. **stod** kann als Abkürzung von "str to double" gelesen werden. Der Funktion wird ein **string** übergeben und sie liefert einen **double**-Wert zurück. Zur Erinnerung: xState.str() liefert einen str der den Text des zuletzt erkannten Tokens enthält. So macht **stod** z.B. aus dem Text "123.45" den Zahlenwert 123.45.

7.3.6 Rückgabewerte

Die Zwischenresultate des Rechners wurden bisher über Referenzvariablen aus den aufgerufenen Produktionen erhalten. Referenzvariablen haben den Vorteil, dass sie in beliebiger Zahl verwendet werden können. Beim Rechnerbeispiel gibt es jedoch jeweils nur einzelne Zwischenresultate.

Das Rechner-Beispiel mit Verwendung von Rückgabewerten befindet sich in dem Verzeichnis:

\TextTransformer\Beispiele\Rechner2

Die Startregel ist kürzer:

```
{{out << }} Expression
{{out << endl;}}
```

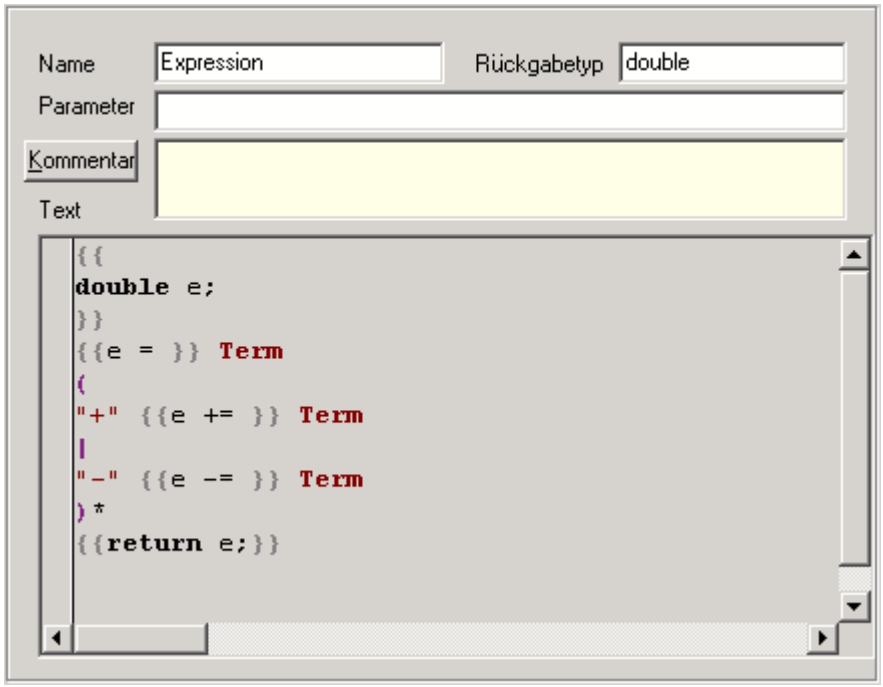
Die Ausgabeanweisung der ersten Zeile hat hier die Besonderheit, dass sie gemäß der reinen Syntax von C++ unvollständig wäre. Im TextTransformer ist diese Schreibweise erlaubt, wenn unmittelbar hinter dem Shift-Operator "<<" die schließende Klammer der semantischen Aktion und der Aufruf einer Produktion folgen. Die Produktion muss einen Wert zurückgeben (s.u.), der in die Ausgabe geschrieben werden kann.

Die abschließende Aktion ist:

```
out << endl;
```

endl eine andere Schreibweise für "\n", d.h. für einen Zeilenumbruch.

Die *Expression*-Produktion hat nun keinen Parameter mehr, dafür aber den **Rückgabetyp: double**



Eine Produktion (oder Token) mit Rückgabetyp, muss auch einen Wert dieses Typs zurückgeben. Dies geschieht in der letzten Zeile:

```
{{return e;}}
```

Die **double**-Variable **e** ist am Beginn der Produktion auf die nun schon bekannte Weise deklariert worden. Den Wert erhält die Variable ihrerseits als Rückgabe der *Term*-Produktion. Dies geschieht in den drei Zeilen, die ebenso wie die obige Ausgabeanweisung, verkürzte C++-Anweisungen sind:

```
{{e = }} Term
{{e += }} Term
{{e -= }} Term
```

Die *Term*- und die *Factor*-Produktion sind entsprechend der *Expression*-Produktion so umgeschrieben worden, dass sie statt mit einem Referenz-Parameter mit einem Rückgabewert arbeiten.

In der ***Number***-Produktion kann die Deklaration einer temporären Variablen eingespart werden, indem der Rückgabewert der **stod**-Funktion direkt weitergereicht wird:

```
number
  {{ return stod(xState.str()); }}
```

7.4 Textstatistik

Die wichtigsten Bedienungselemente von TETRA sollten bekannt sein.

Problemstellung:

Die Anzahl der Zeilen, Zeichen, Worte und Sätze eines Textes sollen ermittelt werden.

TETRA Programm:

Das Projekt befindet sich in dem Ordner:

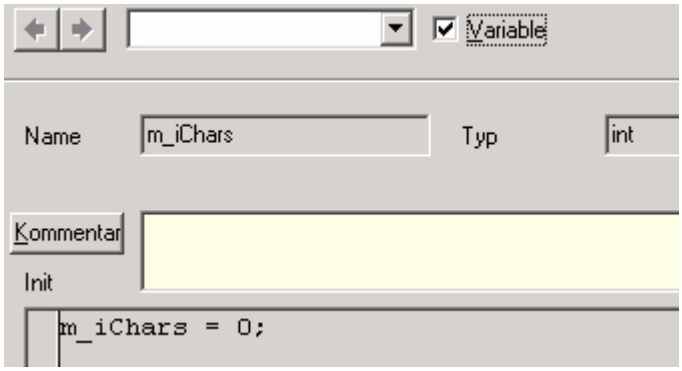
\TextTransformer\Beispiele\TextStats

Dies Projekt demonstriert den Gebrauch von Klassenvariablen und -funktionen. Neben rein linguistischen Anwendungsmöglichkeiten hat es einen praktischen Nutzen, wenn Platzvorgaben für Texte eingehalten werden müssen, wie es z.B. bei der Anmeldung von Sharewareprogrammen bei entsprechenden Verteilern der Fall ist.

7.4.1 Klassenelemente

In den bisherigen Beispielen wurden **lokale Variablen** verwendet, die innerhalb von Aktionen deklariert wurden und bei Bedarf als Parameter an andere Produktionen oder Token weitergereicht wurden. In diesem Projekt werden dagegen **Klassenvariablen** verwendet. Die Deklaration von Klassenvariablen erfolgt auf einer eigens dafür eingerichteten Seite der TETRA-Umgebung. Auf derartig deklarierte Variablen kann dann innerhalb der Aktionen jeder Produktion und jedes Tokens (und jeder Klassenfunktion s.u.) unmittelbar zugegriffen werden. Ihr Wert kann dort verändert werden und steht dann unmittelbar auch in anderen Programmteilen zur Verfügung.
Im aktuellen Projekt gibt es beispielsweise eine Integer-Klassenvariable *m_iChars*, die die Anzahl der bisher gezählten Zeichen speichert. In jeder Token-Aktion wird dieser Wert um die Anzahl der durch das Token erkannten Zeichen erhöht.

Die Deklaration von *m_iChars* erfolgt auf der Seite für Klassenelemente.



Um eine dort eine Variable zu deklarieren, muss in der entsprechenden Checkbox der

Werkzeugleiste ein Häkchen gesetzt werden. Andernfalls würde *m_iChars* als Name einer Klassenfunktion interpretiert.
Der Typ der Variablen ist im Typ-Feld als *int* gesetzt.
Das Textfeld ist nun mit *Init* überschrieben. Für Variablen kann es leer bleiben, es kann aber auch verwendet werden, um der Variablen einen Anfangswert zuzuweisen, den sie erhält, bevor ein neuer Text Transformiert wird. Hier erhält *m_iChars* den Wert 0. Dies ist der default Wert den sie ohnehin erhalten hätte; die explizite Initialisierung an dieser Stelle ist jedoch ein sauberer Programmierstil.
Beim Export von Code für eine Parserklasse, könnte eine fehlende Initialisierung zu Programmfehlern führen, da in C++ eine Integer-variable nicht automatisch mit einem Anfangswert versehen wird.

Andere Klassenvariablen zum Zählen der Anzahl der Zeilen, Worte etc werden auf die gleiche Weise deklariert wie *m_iChars*.

Ein Sonderfall ist die Variable *m_mAbbr*. Sie ist vom Typ mstrstr und wird mit einer Reihe von Ausdrücken initialisiert, die gefolgt von einem Punkt eine Abkürzung bilden.

```
m_mAbbr["am"] = "";
m_mAbbr["Am"] = "";
m_mAbbr["usw"] = "";
m_mAbbr["etc"] = "";
...
```

Diese Liste wird beim Zählen der Sätze eines Textes gebraucht, um unterscheiden zu können, ob ein Punkt ein Satzende markiert oder zu einer Abkürzung gehört. Da die Liste keineswegs vollständig ist, wird die Satzzählung stets mit einem möglichen Fehler behaftet bleiben.
Als Wert wird den Listenelementen stets ein leerer String zugewiesen. Er wird nicht gebraucht. In diesem Projekt wird lediglich die Eigenschaft einer Map ausgenutzt, dass eine Suche nach einem Schlüsselwort in ihr sehr schnell erfolgt, da die Schlüsselworte in sortierter Reihenfolge vorliegen.

Weiter gibt es eine Klassenfunktion *PrintResult*:

```
out << "Text statistics:\n";
out << m_iLines << "\tlines\n";
out << m_iChars << "\tcharacters\n";
out << m_iWords << "\twords\n";
out << m_iNumbers << "\tnumbers\n";
out << m_iSentences << "\tsentences\n"
```

Sie sorgt für die formatierte Ausgabe der Zählung am Ende des Programms.

7.4.2 Token

In den Projektoptionen sind alle auszulssenden Zeichen deaktiviert, so dass die Menge der Token sämtliche Teile eines Textes abdecken müssen inclusive der Leerzeichen und Zeilenumbrüche.

Ein Text besteht demnach aus

WORD	Worten
NUMBER	Zahlen

ABBREVIATION	Abkürzungen
CONTINUATION	Folgen von Punkten, wie "..."
LINEFEED	Zeilenumbrüchen
SENTENCE_END	Satzenden (Punkt, Ausrufe- oder Fragezeichen)
SPECIAL_CHAR	übrige Zeichen

In den Aktionen der Token werden jeweils die Zähler aktualisiert. Z.B. für WORD:

```
m_iWords++;
m_iChars += xState.length();
```

Hier wird der Wortzähler um Eins erhöht und der Zeichenzähler um die Anzahl der Zeichen des Wortes.

Etwas komplizierter ist die Aktion des Tokens *ABBREVIATION*: (\w+)\.

```
if(xState.length() > 2 &&
   !m_mAbbr.findKey(xState.str(1)))
  m_iSentences++;

m_iWords++;
m_iChars += xState.length();
```

Besteht der erkannte Text aus nur einem Buchstaben gefolgt von einem Punkt oder wird der Text vor dem Punkt in der Abkürzungsliste gefunden, gilt der erkannte Text als Abkürzung. Andernfalls markiert der Punkt ein Satzende und der Satzzähler wird heraufgesetzt.

7.4.3 Produktionen

Die Startregel des Projekts ist *TextStat*. *TextStat* ruft in einer Schleife die Produktion *Text* auf, die aus den alternativen Token besteht. Beide Produktionen sind sehr einfach und enthalten nichts Neues.

Zusätzlich gibt es noch die Produktion *CountChars*. *CountChars* bietet eine sehr einfache Möglichkeit allein die Anzahl der Zeichen eines Textes zu ermitteln.

7.5 GrepUrls

Die wichtigsten Bedienungselemente von TETRA sollten bekannt sein.

Problemstellung:

In Html-Dateien sollen alle Links gefunden und aufgelistet werden, die ins Web verweisen.

TETRA Programm:

Das Projekt befindet sich in dem Ordner:

\TextTransformer\Beispiele\GrepUrls

Dies Projekt demonstriert den Einsatz des TextTransformers als Grep-Tool. GREP (Global Regular Expression Print) ist ein in der Unix-Welt sehr bekanntes Programm, mit dem nach Textmustern in Dateien gesucht werden kann. Mit Hilfe Transformations-Mangers ist eine solche Aufgabenstellung auch vom TextTransformer zu bewältigen. Zugleich demonstiert dieses Beispiel, wie die gefundenen Informationen in Containern gesammelt und schließlich formatiert ausgegeben werden können.

7.5.1 Produktionen

Schaut man sich eine Html-Datei nicht im Browser sondern als Text an, so ist z.B. folgendes ein Link auf die TextTransformer Seiten:

<a href="http://www.texttransformer">

Wenn der Text innerhalb der Anführungszeichen - die URL - mit "http://www." beginnt, so verweist der Link auf externe Seiten. Genau solche Verweise sollen in diesem Beispiel aufgespürt werden. Mit nur zwei einfachen Produktionen lässt sich das ziel erreichen::

```
GrepUrls ::=

  (
    Url
  | SKIP
  )*

Url ::=

  "<a href=\"http://www."
   SKIP
  "\""
```

Eine Html-Seite besteht aus den gesuchten Url's und anderem Text. Die gesuchte Url wird eingeleitet durch "<a href=\"http://www." und erstreckt sich bis zum Schließen der Anführung.

In den Projektoptionen wird die Beachtung der Groß-/Kleinschreibung abgeschaltet, da diese in HTML-Dateien ignoriert wird. So wird dann z.B. auch das Tag

```
  "<A HREF=\"http://www."
```

gefunden.

7.5.2 Klassen-Variablen und -methoden

Nun sollen einige Variablen und Methoden vorgestellt werden, die auf der Element-Seite definiert sind.

Die Fundstellen der Url's sollen in einer Klassenvariablen vom Typ *mstrstr* gesammelt werden.

```
mstrstr m_mUrl
```

Als Schlüssel soll dabei die gefundene Url selbst dienen, und der Wert wird aus dem Namen der aktuellen Datei und der zugehörigen Zeilennummer zusammengesetzt. Diese erhält man aus Methoden des Parsers:

SourceName() Name der aktuellen Quelldatei
xState.Line() Zeilennummer

Eine Variable vom Typ format soll helfen, die Namen und Nummern zu einem Gesamtausdruck zu verknüpfen:

```
format m_fPosition
```

Durch die Anweisung:

```
m_fPosition.parse("  Page: %|1$|%|50t|Line: %|2$|");
```

wird m_fPosition mit dem Format-String " Page: %|1$|%|50t|Line: %|2$|" initialisiert.
Hierin steht "%|1$|" für die Position des ersten Arguments, %|2$| für die Position des zweiten Arguments und %|50t| für das Auffüllen mit Leerzeichen des davorstehenden Textteiles auf insgesamt 50 Zeichen. Wenn z.B. der Dateiname "D:\C_biblio\boost\index.htm" und die Zeilennummer *52* durch den %-Operator an das Format-Objekt übergeben wurden, gibt dieses durch Aufruf der str()-Methode den folgenden String zurück:

```
"  Page: D:\C_biblio\boost\index.htm             Line: 52"
```

Um die Dateinamen nicht zu lang werden zu lassen, sollen sie in der kürzeren Form mit relativen Pfadangaben ausgedruckt werden. Im Projekt gibt es deshalb eine spezielle Methode, die die absoluten Pfade, die SourceName() liefert in die relativen Pfade umformt:

```
str GetRelPath()
{
  return ".." + SourceName().substr(SourceRoot().length());
}
```

Die Funktion besteht aus nur einer, schwer lesbaren Zeile. Was in dieser Zeile passiert, wird klarer, wenn man sie in mehrere Teilschritte aufspalten würde:

```
str sRoot = SourceRoot();
```

```
unsigned int pos = sRoot.length();
str sAbsFilename = SourceName();
str sPart = sAbsFilename.substr(pos);
str sRelFilename = ".." + sPart;
```

Der von der Methode zurückgegebene relative Dateiname wird zusammengesetzt aus dem String ".." und dem Teilstring des absoluten Dateinamens, der dem String, der das Quellverzeichnis bezeichnet, folgt. Bezogen auf "D:\C_biblio\boost\index.htm" mit dem Quellverzeichnis "D:\C_biblio\boost" erhält man so:

..\index.htm

Eine weitere auf der Seite für Klassenelemente definierte Klassenmethode ist *AddPosition*, der eine Url *xsUrl* und die dazugehörige Fundstelle *xsWhere* als Parameter übergeben werden:

```
{{
if(m_mUrl.findKey(xsUrl))
{
  m_mUrl[xsUrl] += "\n" + xsWhere;
}
else
  m_mUrl[xsUrl] = xsWhere;
}}
```

Sie berücksichtigt, dass es eventuell mehrere Fundstellen für die gleiche Url geben kann.In der Funktion wird zunächst abgefragt, ob eine Adresse bereits einmal aufgefunden wurde. Falls nein, wird im else-Zweig die aktuelle Fundstelle als Wert des Url-Schlüssels gesetzt. Falls ja, wird die neue Fundstelle einfach an die bisherigen angefügt.

Vor Beendigung des Programms wird m_mUrl mittels der Funktion *PrintAll* ausgedruckt, wobei die gefundenen Internet-Adressen automatisch in alphabetischer Ordnung erscheinen:

```
m_mUrl.reset();
while(m_mUrl.gotoNext())
{
  out << m_mUrl.key() << endl;
  out << m_mUrl.value() << endl << endl;
}
```

7.5.3 Zusammenfassung

Der vollständige Code der Url-Produktion ist nun:

```
"<a href=\"http://www."
 SKIP
{{
m_fPosition % GetRelPath() % xState.Line();
```

```
AddPosition(xState.str(), m_fPosition.str());
}}
"\""
```

Nachdem der Text der Url durch SKIP erkannt wurde, wird er zusammen mit der Information über die Fundstelle an die Klassenfunktion *AddPosition* übergeben.

Und schließlich die vollständige *GrepUrls*-Produktion:

```
(
  Url
| SKIP
)*
{{
if ( IsLastFile() )
  PrintAll();
}}
```

Hier ist noch ein wichtiger Punkt zu betrachten: Das Programmm soll die sortierte Liste der Internetadressen ausgeben, die in allen Html-Dateien gefunden wurden. Die Sortierung ist aber nur möglich, nachdem **alle** Dateien durchsucht wurden. Deshalb wird die *PrintAll*-Funktion nur ausgeführt, wenn diese Bedingung erfüllt ist. Ob sie erfüllt ist, kann mit der Methode *IsLastFile* des Parser-Zustands ermittelt werden. *IsLastFile* gibt nur dann den Wert true zurück, wenn keine weitere Datei mehr folgt.

7.5.4 Verzeichnis durchsuchen

Mit dem fertigen Projekt können nun in gewohnter Weise eine einzelne Html-Datei verarbeitet werden, indem sie in das Eingabefenster des TextTransformers geladen werden. Hier soll aber demonstriert werden, dass es mit dem TextTransformer auch möglich ist ganze Verzeichnisse zu verarbeiten. Im gegenwärtigen Falle wären dies sämtliche Dokumente eine Web-Site.

Hierzu wird der Transformations-Manager aufgerufen. Am bequemsten geht dies mit dem Schalter

Sollte das Projekt noch nicht kompiliert sein, so geschieht dies nun automatisch, bevor der Transformations-Manager-Dialog geöffnet wird. Die Arbeit mit dem Transformations-Manager wird ausführlich in einem eigenen Kapitel dieser Hilfe dargestellt. Hier werden nur die für diese Beipiel wichtigen Schritte zur Erstellung eines Managements kurz dargestellt.

Das Management befindet sich auch fertig in:

C:\Programme\TextTransformer\Beispiele\GrepUrls\GrepUrls.ttm

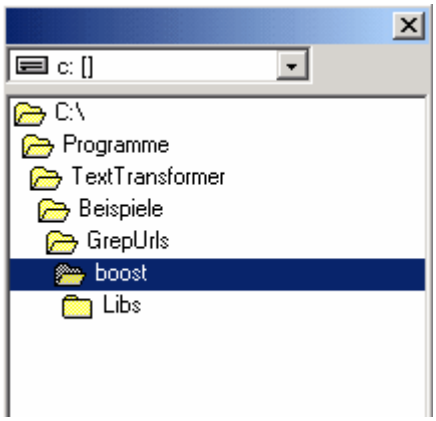
Wenn Sie auch in Zukunft beabsichtigen HTML-Seiten zu bearbeiten, so empfiehlt es sich gleich jetzt einen Filter für diesen Dateityp zu definieren, der dann immer wieder bei der Auswahl von Quelldateien im TextTransformer benutzt werden kann.

Zunächst wird das Verzeichnis ausgewählt, in dem sich die zu transformierenden Dateien befinden.

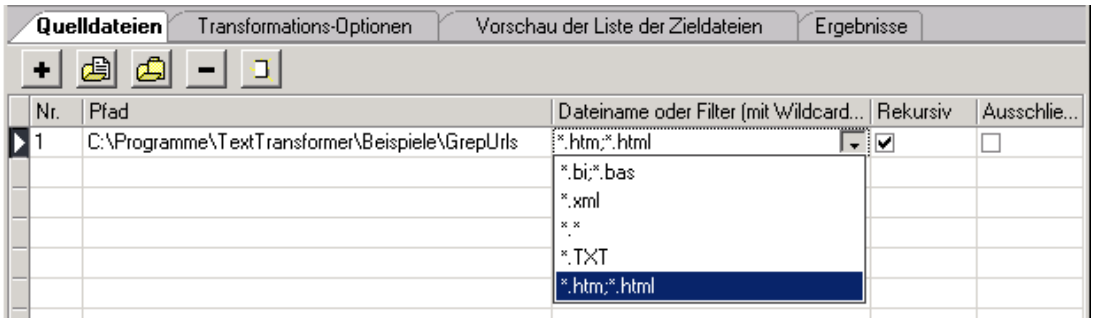
Für Diese Beispiel wurden einige der Html-Dateien, die zur Web-Site von boost gehören in ein Unterverzeichnis von GrepUrl kopiert:

Über den Schalter

lässt sich ein Dialog öffnen, in dem des Quellverzeichnis ausgewählt wird.



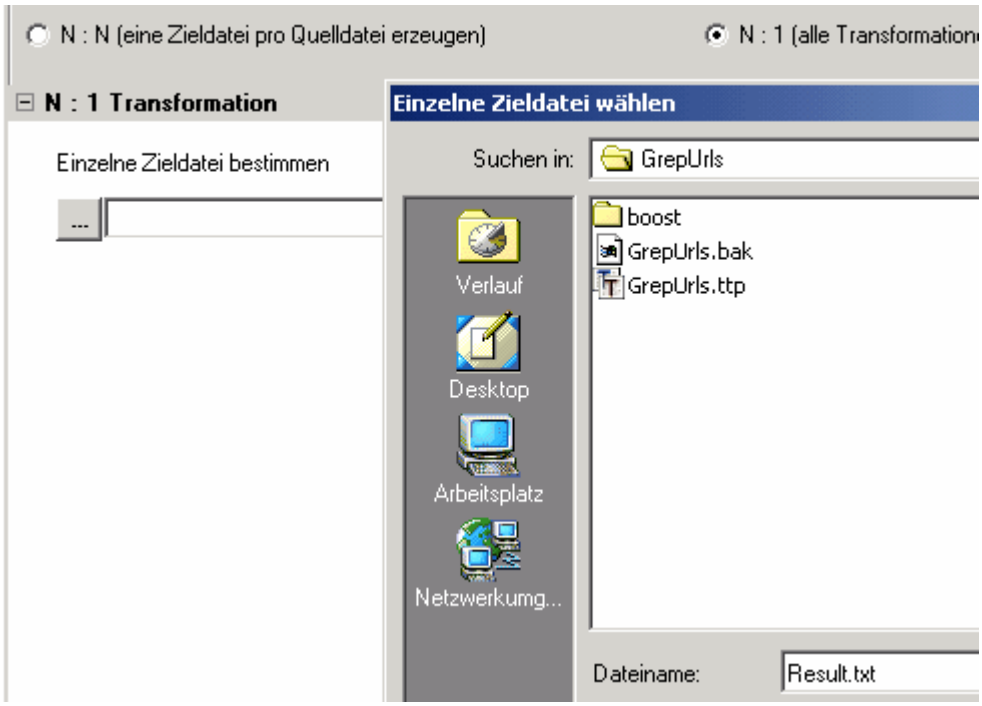
Nachdem die Auswahl bestätigt wurde, wird eine neue Zeile in die Tabelle der Quelldateien/Verzeichnisse eingefügt. In dieser Zeile kann nun das Kästchen *Rekursiv* aktiviert werden, um auch die Dateien der Unterordner zu durchsuchen. Wenn, wie oben empfohlen der Filter für HTML-Dateien definiert wurde, kann dieser nun in der Auswahlbox der Spalte *Dateiname oder Filter* ausgewählt werden. Der Filter "*.htm;*.html" kann aber auch direkt in das Feld geschrieben werden.



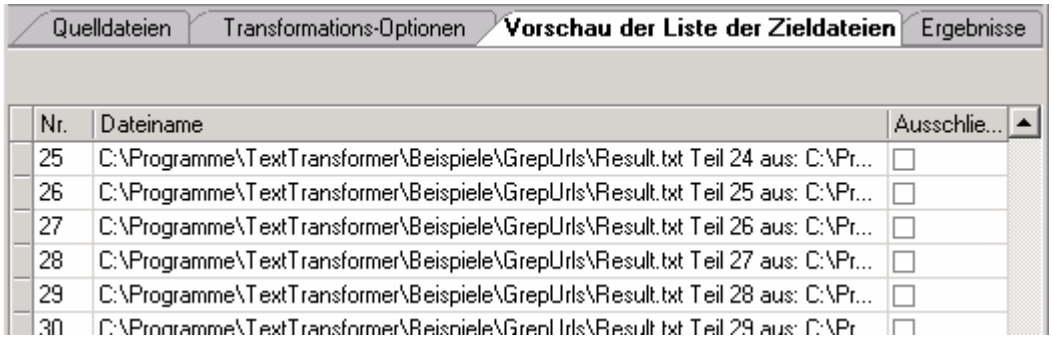
Auf der Seite *Transformations-Optionen* der Transformations-Managers muss nun eingestellt werden, dass eine N:1 Transformation geplant ist, d.h. dass die Ergebnisse der Transformationen aller Quelldateien in nur einer einzigen Zieldatei zusammengefasst werden sollen.
Mit dem Schalters

wird ein Dateiauswahl-Dialog geöffnet, mit dem zum gewünschten Verzeichnis navigiert werden kann, in dem dann entweder eine bereits existierende Datei ausgewählt wird, oder ein zu

erzeugende Datei als dateiname angegeben wird.



Auf der nächsten Seite der Transformations-Managers wird nun eine Vorausschau agezeigt, in der es für jede Quelldatei in einer Tabelle eine Zeile gibt, die ihren Beitrag zu der zu erzeugenden Datei markiert.



Nun kann die Adresssuche gestartet werden:

Nach einem Doppelkick auf eine beliebige Zeile in der Tabelle auf der *Resultate-Seite*, wird der Transformations-Manager geschlossen und der Ergebnis-Text wird im Ausgabefenster der IDE angezeigt.

7.6 BinaryCheck

Die wichtigsten Bedienungselemente von TETRA sollten bekannt sein.

Problemstellung:

Es soll getestet werden, ob eine Datei binäre Daten enthält.

TETRA Programm:

Das Projekt befindet sich in dem Ordner:

\TextTransformer\Beispiele\BinaryCheck

Das Beispiel demonstriert den denkbar einfachsten Parser für binäre Dateien und verwendet eine Vorausschau-Produktion.

Das Projekt kann als Präprozessor für andere Projekte verwendet werden, der sicher stellt, dass die bearbeite Datei nicht binär ist.

7.6.1 Vorausschau

Das Projekt besteht aus nur zwei Produktionen. Die eine, *IsBinary,* ist extrem simpel:

```
SKIP? NULL
```

Sie kann einen String nur erfolgreich parsen, wenn er mit einer binären Null endet.

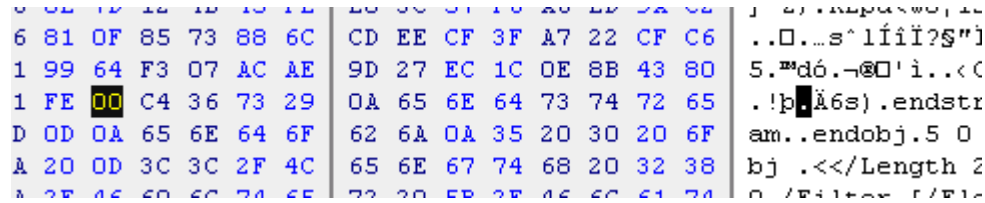
Diese Produktion wird innerhalb der Produktion *BinaryCheck* zur Vorausschau verwendet:

```
IF(!IsBinary())
  SKIP? {{ out << xState.str();  }}
END
```

Syntaktisch wird der Unterschied der Verwendung als Vorausschau zu einem normalen Aufruf einer Produktion durch das angehängte Klammerpaar "( )" innerhalb der IF-Klammern gekennzeichnet. Bei der Vorausschau wird die Quelldatei ab der aktuellen Position - hier der Dateianfang - solange geparst, bis entweder die Produktion erfolgreich abgearbeitet ist oder bis ein Fehler auftritt. *IsBinary* ist genau dann erfolgreich, wenn es ein *NULL*-Zeichen in der Datei gibt.

Das kann man im Debugger sehen. Mit dem Öffnen des Projekts sollte die Datei BinaryTest.pdf in den Betrachter geladen worden sein. Im hexadezimalen Modus des Betrachters kann man auch die NULL-Zeichen sehen. Wenn man mit

in die Vorausschau hinein geht und den Schalter mehrmals drückt, ergibt sich schließlich folgendes Bild, in dem das gefundene NULL-Zeichen markiert ist.



Nur, wenn es kein *NULL*-Zeichen gibt wird die Datei hier als Text-Datei betrachtet. (Tatsächlich könnte die Datei dennoch für eine binäre Verwendung gedacht sein.)
Wenn es eine Textdatei ist, wird mit *SKIP?* an das Ende gesprungen und der gesamte Text dann vollständig in die Ausgabe geschrieben.
So kann das Projekt als Präprozessor für andere Projekte verwendet werden, der sicher stellt, dass die Quelldatei nicht binär ist.

Zusätzlich wird am Anfang von *BinaryCheck* noch vor der Vorausschau die Größe der Datei geprüft. Binäre Dateien sind oft sehr groß, da sie Bilddaten enthalten oder gar Sprachaufzeichnungen und Filme. Als Grenzwert werden hier 1000000 Bytes gewählt.

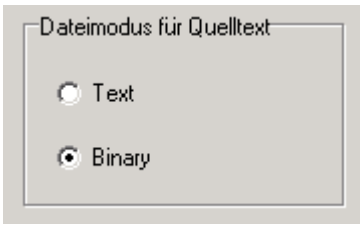
```
{{
int iMaxSize = 1000000;
if( file_size(SourceName()) > iMaxSize )
  throw CTT_Error("file size > " + itos(iMaxSize));
}}
```

Textdateien erreichen selten diese Größe. Ob noch größere Dateien zugelassen werden sollen hängt von der Art der Dateien ab. Auch sollte bedacht werden, dass die Quelldateien zum Parsen vollständig in den Arbeitsspeicher geladen werden.

7.6.2 Als Präprozessor

Ein TextTransformer-Projekte kann als Präprozessor für ein zweites Projekt verwendet werden. D.H. das zweite Projekt übernimmt den Zieltext des ersten als Quelltext. *BinaryCheck* wäre ein geeigneter Präprozessor für Projekte, die auf alle Texte eines Verzeichnisses angewendet werden sollen, ungeachtet ihrer speziellen Dateierweiterungen. Nur für Textdateien liefert BinaryCheck den Quelltext, bei binären Dateien schlägt BinaryCheck fehl und liefert keinen Text.

Nur, wenn die Quelldatei im Binärmodus geöffnet wird, ist garantiert, dass *BinaryCheck* funktioniert. Werden binäre Dateien im Textmodus geöffnet, werden sie meist nur unvollständig eingelesen.
Der Binärmodus lässt sich in den Projektoptionen auf der Kodierungsseite setzen.



Ab TextTransformer 1.5.5 ist der Binärmodus für neue Projekte voreingestellt.

Nun kann *BinaryCheck* in den Projektoptionen eines anderen Projekts als Präprozessor ausgewählt werden. Ein Ausschnitt vom Resultat eines Testlaufs von *TextStat* mit dem *BinaryCheck*-Präprozessor im Transformations-Manager ist unten abgebildet:

	28.08.2008	14:45:32	Starte C:\Program Files\Minimal Website\bin\mini_ws.exe
	28.08.2008	14:45:33	file size > 1000000
	28.08.2008	14:45:33	Fehler bei der Vorverarbeitung mit (Präprozessor): BinaryCheck
	28.08.2008	14:45:33	C:\Program Files\Minimal Website\bin\mini_ws.exe : Transformation nicht erfolgreich
	28.08.2008	14:45:33	Starte C:\Program Files\Minimal Website\bin\readme.txt
	28.08.2008	14:45:33	erfolgreich transformiert : C:\Program Files\Minimal Website\bin\readme.txt -> D:\Text
	28.08.2008	14:45:33	Starte C:\Program Files\Minimal Website\unins000.dat
	28.08.2008	14:45:33	Fehler bei der Vorverarbeitung mit (Präprozessor): BinaryCheck
	28.08.2008	14:45:33	C:\Program Files\Minimal Website\unins000.dat : Transformation nicht erfolgreich

Man sieht, dass die Transformation von *mini_ws.exe* an seiner Größe gescheitert ist und *unins000.dat* als Binärdatei identifiziert wurde. Die Textdatei *readme.txt* hingegen wurde korrekt verarbeitet.

7.7 E-Mail-Adresse

Die wichtigsten Bedienungselemente von TETRA sollten bekannt sein.

Problemstellung:

Eine E-Mail-Adresse soll gemäß dem komplexen RFC 822 Standard analysiert werden. Gemäß diesem Standard sind nicht nur einfache Adressen wie

dme@TextTransformer.de

zulässig, sondern auch Konstrukte wie:

Detlef Meyer-Eltz <dme@TextTransformer.de (Parsergenerator) >

TETRA Programm:

Das Projekt befindet sich in dem Ordner:

\TextTransformer\Beispiele\Mailbox

Das Projekt demonstriert, wie ein TextTransformer-Programm aus einer vorhandenen Syntax-Spezifikation erzeugt werden kann. Dabei wird auch gezeigt, wie ein in der Spezifikation verborgener LL(1)-Konflikt aufgelöst wird.

Einen vollständigen MIME-Parser für E-Mails gibt es unter:

http://www.texttransformer.org

7.7.1 Syntax-Spezifikation

Folgende Syntax-Spezifikation befindet sich ist in dem Buch: J.E.F. Friedl: Reguläre Ausdrücke, O'Reilly, 1998.

	Element	Beschreibung
1	mailbox	addr-spez | phrase route-addr
2	addr-spec	local-part @ domain
3	phrase	(word)+
4	route-addr	< (route)? addr-spez >
5	local-part	word (. word)*
6	domain	sub-domain (. sub-domain)*
7	word	atom | quoted-string
8	route	@ domain (, @ domain)* :
9	sub-domain	domain-ref | domain-literal
10	atom	(ein beliebiges Zeichen außer specials, space oder ctl)+
11	quoted-string	" (qtext | quoted-pair)* "
12	domain-ref	atom
13	domain-literal	(dtext | quoted-pair)*
14	char	Ein ASCII-Zeichen (oktal 000-177)
15	ctl	Ein ASCII-Kontrollzeichen (oktal 000-037)
16	space	Leerzeichen (ASCII 040)
17	CR	Carriage Return (Wagenrücklauf, ASCII 015)
18	specials	Eins der Zeichen ()<>@,;:\\"\.\[\]
19	qtext	Ein char außer *****, **** oder CR
20	dtext	Ein char außer [,], \ oder CR
21	quoted -pair	\ char
22	comment	**(** (ctext | quoted-pair | comment)* **)**
23	ctext	Ein char außer '(', ')',' \' oder CR

Friedl konstruiert aus dieser Beschreibung einen einzigen regulären Ausdruck, der aus 4724 Zeichen besteht. Der folgende Nachbau der Grammatik im TextTransformer ist in etwa so lang, wie die Spezifikation selbst.

7.7.2 Produktionen und Token

Glücklicherweise ist die Syntax der Spezifikation der des TextTransformers sehr ähnlich. Die Elemente können leicht in Produktionen und Token umgewandelt werden. Da Bindestriche in Skript-Namen des TextTransformers nicht zulässig sind werden sie in Unterstriche überführt. Zugleich werden Einzelzeichen in Anführungszeichen eingeschlossen. Elemente, die aus einzelnen

Zeichen bestehen werden zu Token. Ihre Namen werden zur Abhebung von den Produktionen durchgängig groß geschrieben.

	Produktion	Definition
1	mailbox	addr-spez | phrase route_addr
2	addr_spec	local_part "@" domain
3	phrase	(word)+
4	route_addr	< (route)? addr_spez >
5	local_part	word ("." word)*
6	domain	sub_domain ("." sub_domain)*
7	word	ATOM | quoted_string
8	route	"@" domain ("," "@" domain)* :
9	sub_domain	domain_ref | domain_literal
11	quoted_string	"\"" (QTEXT | QUOTED_PAIR)* "\""
12	domain_ref	ATOM
13	domain_literal	(DTEXT | QUOTED_PAIR)*

	Token	Definition
10	ATOM	[^()<>@,;:\\"\.\[\]\x{00}-\x{20}\x{7f}]+
14	CHAR	[\x00-\x7F]
17	CR	\r
18	SPECIALS	[()<>@,;:\\"\.\[\]]
19	QTEXT	[^*\\\r\x80-\xFF]
20	DTEXT	[^\[\]\\\r\x80-\xFF]
21	QUOTED_PAIR	\\[\x00-\x7F]
23	CTEXT	[^()\\\r\x80-\xFF]

Die comment-Produktion wird in den Projekteinstellungen als Einschluss gesetzt. Dadurch wird automatisch vor jedem neuen Token getestet, ob möglicherweise ein Kommentar eingeschoben ist. Solche Kommentare können verschachtelt sein im Gegensatz zur Verwendung regulärer Ausdrücke zur Kommentarerkennung.

Das komplette Projekt befindet sich in:

\TextTransformer\Beispiele\Mailbox\mailbox1.ttp

7.7.3 Konflikt erkennen

Soweit ging alles ohne Probleme und das Projekt scheint bereits fertig zu sein. Wird nun aber die Startregel *mailbox* ausgewählt und geparst erscheinen zwei Fehlermeldungen:

mailbox: LL(1) Fehler: "\"" ist Anfänger mehrerer Alternativen

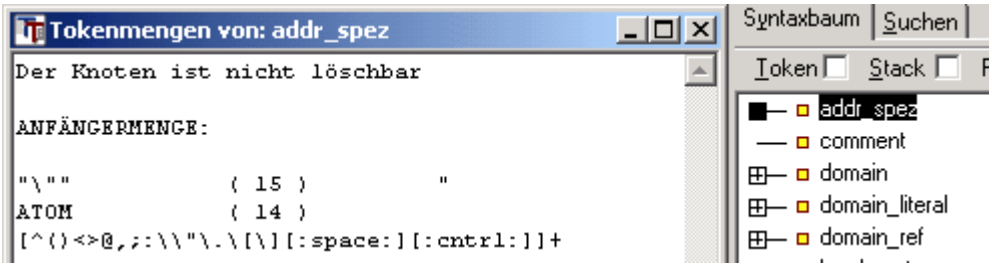
mailbox: LL(1) Fehler: "ATOM" ist Anfänger mehrerer Alternativen

Wenn ein derartiger LL(1)-Konflikt vorliegt, kann ein Parser anhand des aktuellen Textes nicht entscheiden, welche Regel er als nächste anwenden soll.
Der TextTransformer - im Gegensatz zu einigen anderen verwandten Tools - hilft derartige Fehler zu finden. Ihre Beseitigung erfordert jedoch ein Verständnis der zu entwickelnden Grammatik und einige Kombinationsgabe.

In der *mailbox* Produktion:

```
addr_spez | phrase route_addr
```

beginnen die Alternativen mit *addr_spez* und *phrase*. Wenn *addr_spez* im Navigationsbaum auf der rechten Seite der TextTransformer-Oberfläche ausgewählt wird und dann mit der rechten Maustaste das Popup-Menu aufgerufen wird, kann man sich die Anfängermenge dieser Produktion anzeigen lassen:



Bei der *phrase* Produktion ergibt sich entsprechend genau die gleiche Anfängermenge.
Schaut man sich die Definitionen dieser Produktionen an, ist die Ursache des Konflikt zunächst noch nicht zu erkannen:

```
addr_spez ::= local_part "@" domain

phrase ::= word+
```

Steigt man jedoch eine weitere Ebene hinab wird der Konflikt offenbar:

```
local_part ::= word ("." word)*

word ::= ATOM | quoted_string
```

Indirect beginnen also sowohl *addr_spez* als auch *phrase* mit word.

7.7.4 Konflikt auflösen

Um den Konflikt aufzulösen, muss *word* gewissermaßen ausgeklammert werden.
Für *phrase* ist diese einfach zu machen. Die Regel wird neu definiert als:

```
phrase = word*
```

und alle Stellen, an denen *phrase* in der Grammatik vorkommt werden ersetzt durch

```
word phrase
```

Die *mailbox*-Produktion ist gerade die einzige Stelle an der *phrase* vorkommt; sie wird somit zu:

```
mailbox ::= addr_spez | word phrase route_addr
```

Analog wird mit *local_part* verfahren. Die Produktion wird umdefiniert zu

```
local_part ::= ("." word)*
```

und alle Stellen, an denen *local_part* in der Grammatik vorkommt werden ersetzt durch

```
word local_part
```

Damit wird addr_spez zu:

```
addr_spez ::= word local_part "@" domain
```

Schießlich wird *word* auch aus *addr_spez* herausgezogen, womit diese Produktion ihre alte Form zurückerhält (jedoch mit anders definierter Produktion *local_part*):

```
addr_spez ::= local_part "@" domain
```

Hier muss aufgepasst werden, da *addr_spez* auch in *route_addr* verwendet wird. Damit wird letztere zu:

```
route_addr ::= "<" (route)? word addr_spez ">"
```

Und schließlich

```
mailbox ::= word addr_spez | word phrase route_addr
```

Nun kann die Ausklammerung von *word* vollendet werden:

```
mailbox ::= word ( addr_spez | phrase route_addr )
```

Die Kompilierung der Regel ergibt nun keinen Konflikt mehr.

Das so korrigierte Projekt befindet sich in:

\TextTransformer\Beispiele\Mailbox\mailbox2.ttp

7.8 Guard

Die wichtigsten Bedienungselemente von TETRA sollten bekannt sein.

Hier soll vor allem die Verwendung von Unterausdrücken regulärer Ausdrücke demonstriert werden.
Dies Beispiel ist vor allem für C++-Programmierer interessant, aber auch ohne den inhaltlichen Zweck des Programms zu verstehen sind die verwendeten Methoden für Nicht-Programmierer lehrreich.

Problemstellung:

In einen C++-Quelltext sollen an den Ein- und Austrittspositionen von Klassenmethoden - d.h. nach der öffnenden und vor der schließenden Klammer des Funktionskörpers - zusätzliche Befehle eingefügt werden, die dem Auffinden oder Vermeiden von Fehlern oder dem Messen der Geschwindigkeit der Funktionen dienen können. Dazu soll den eingefügten Befehlen der Name der Klasse und der Name der Methode als Parameter übergeben werden.
Wird beispielsweise im Quelltext eine Methode definiert:

```
void CClass::Name ( int xi )
{
   ...
}
```

soll im Zieltext stehen:

```
void CClass::Name ( int xi )
{
    CGuard G("CClass", "Name");
   ...
    G.stop();
}
```

Die Klasse CGuard ("guard" ist das englische Wort für "Wächter") kann im Konstruktor bestimmte Ressourcen blockieren oder eine Zeitmessung anstoßen und im Destruktor die Ressource wieder freigeben bzw. die Zeit stoppen.
Eine weiter Möglichkeit wäre eine Transformation obiger Methoden zu:

```
void CClass::Name ( int xi )
{
    try
    {
       ...
    }
    catch(...)
    {
        throw CException("CClass::Name");
    }
}
```

TETRA Programm:

Das Beispiel *guard* befindet sich in:

\TextTransformer\Beispiele\Guard\guard.ttp

Das TETRA-Programm dieses Beispiels dient nur Demonstrationszwecken. Es ist nicht garantiert, dass damit jede C++-Source-Datei zu transformieren wäre. Immerhin soll es den Code verarbeiten können, den TETRA selbst erzeugt. Als Beispielstext dienen daher Parser, die vom TextTransformer generiert wurde:

\TextTransformer\Beipiele\Guard\rechnerparser.cpp
\TextTransformer\Beipiele\Guard\guardparser.cpp

7.8.1 Startregel: guard

Die Startregel des Programms trägt den Namen des Projekts guard. Sie soll den gesamten Text einer von TETRA erzeugten C++-Quelldatei erkennen können:

```
(
  SKIP                {{ out << xState.copy(); }}
| constructor
| destructor
| member_function
| global_declaration  {{ out << xState.lp_copy(); }}
| LINE_COMMENT        {{ out << xState.copy(); }}
| PREPROCESSED        {{ out << xState.copy(); }}
| USING               {{ out << xState.copy(); }}
)+
```

Die Namen der alternativen Productionen, die in der Schleife aufgerufen werden können, bezeichnen die Strukturen, die mit diesen Regeln erfasst werden sollen. Ein C++-Quelltext besteht demnach aus Konstruktoren, Destruktoren, Klassenfunktionen, Kommentaren, Zeilen, die vom Präprozessor verarbeitet werden, using-Direktiven und einem undefinierten Rest, der durch das SKIP-Symbol abgedeckt wird.
Die Konstruktoren, Destruktoren und Klassenfunktionen sollen mit "Wächtern" instrumentiert werden. Die anderen Teile des C++-Codes sollen unverändet in den Zieltext kopiert werden.

7.8.2 Quelltext kopieren

Die Methode zum Kopieren des Textes, der vom letzten Token erkannt wurde, ist schon bekannt. Die Leerzeichen vor dem Text soll in dieser Kopie eingeschlossen sein. D.h. es wird der Text vom Ende des vorletzten Tokens bis zum Anfang des nächsten erwarteten Tokens kopiert:

```
{{ out << xState.copy(); }}
```

Dieser Befehl ist eine abgekürzte Schreibweise für:

```
{{ out << xState.str(-1) << xState.str(); }}
```

wo auf den ausgelassenen Text mit ***xState**.str(-1)* und auf den erkannten Text mit ***xState**.str()* zugegriffen wird. Durch die Angabe eines Parameters kann auf bestimmte Abschnitte des Eingabetextes relativ zum aktuell erkannten Textabschnitt zugegriffen werden. Der Parameter "-1" hat die besondere Bedeutung, dass er sich auf die bei der letzten Erkennung übersprungenen auszulassenden Zeichen bezieht. xState.str(-1) gibt den Text zurück, der sich zwischen dem Ende des zuvor erkannten Tokens und dem Beginn des zuletzt erkannten Tokens befindet.
Bei den Beispielen "Atari" und "Wortvertauschung" gab es keine ausgelassenen Zeichen. Es gab nur die vom SKIP-Symbol erfassten Textteile und die ersetzten Textteile. Im gegenwärtigen Beispiel gilt die Standardeinstellung der Projektoptionen, wonach Zeilenumbrüche, Leerzeichen und Tabulatoren auszulassen sind. Durch den Aufruf von ***xState**.copy()* nach jedem erkannten Token wird sichergestellt, dass der Originaltext vollständig inclusive der auszulassenden Zeichen in den Zieltext kopiert wird.

Neu ist hier eine analoge Methode zum Kopieren des gesamten Textes, der durch eine Produktion erkannt wurde.

```
{{ out << xState.lp_copy(); }}
```

Das "lp" von "lp_copy" steht für "letzte Produktion".

7.8.3 Token

Neben einigen direkt in den Regeln definierten literalen Token (und dem STRING-Token s.u.) gibt in diesem Beispiel zwei Gruppen von Token. Die erste besteht aus den Token:

```
LINE_COMMENT        //[^\r\n]*
PREPROCESSED        #[^\r\n]*
USING               using [^\r\n]*
```

Sie beginnen unterschiedlich und enden mit **[^\r\n]***. Letzter Ausdruck beschreibt eine beliebig häufige Wiederholung von Nicht-Zeilenende-Zeichen. Insgesamt bezeichnen die Ausdrücke also Textabschnitte die mit "//" or "#"' oder "using" beginnen und sich bis zu den Zeilenenden erstrecken. Ein C++Programmierer erkennt sofort dass mit diesen Ausdrücken Zeilenkommentare, Präprozessordirektiven und using-Direktiven abgedeckt sind.

Die **zweite Gruppe von Token** besteht aus den Token:

DECLARATOR
DESTRUCTOR

Sie beginnen jeweils mit einem Ausdruck ähnlich dem folgenden:

```
(((\w+::)*\w+)::)?(\w+)
```

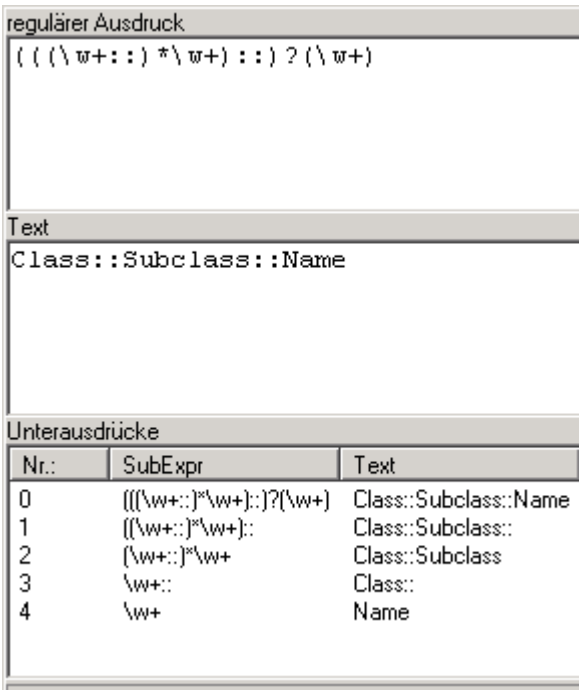
Dieser Ausdruck scheint unnötig kompliziert. Der einfachere Ausdruck:

```
(\w+::)*\w+
```

würde ebenso wie der erste Texte der Art:

Name
Class::Name
Class::Subclass::Name
usw.

erkennen, also Namen und Klassenmethoden.
Die komplizierte Form des Ausdrucks ermöglicht den Zugriffs auf Unterausdrücke. Jeder Klammerausdruck deckt einen Teil des insgesamt vom regulären Ausdruck erkannten Textes ab und dieser Teil kann im Interpreter ermittelt werden. Welcher Text welchem Klammerausdruck zugeordnet ist lässt sich am einfachsten mit Hilfe eines Tools erkennen, das im Hilfemenü unter **Regex Test** aufrufbar ist:



Aus der unteren Tabelle ist zu ersehen, dass der Unterausdruck mit dem Index 2 den Scope und der Unterausdruck mit dem Index 4 den Namen einer Klassenmethode enthält. Dies wird im *guard* -Projekt ausgenutzt um beide Strings in getrennte Klassenvariablen zu schreiben.

```
{{
m_sScope = xState.str(2);
m_sName = xState.str(4);
}}
```

Diese Variablen sind auf der Interpreter-Seite der IDE definiert. Sie werden in den später beschriebenen Funktionen *print_at_enter* und *print_at_exit* gebraucht.

Die vollständigen Definitionen sind:

DESTRUCTOR ::= (((\w+::)*\w+)::)(~\w+)

DECLARATOR ::=
(((\w+::)*\w+)::)(\w+) \//Scope(s) und Name, z.B.: CSub::CClass::Func

```
\s* \// optionale Leerzeichen
\([^)]*\) // Parameter, z.B.: ( int xi )
```

Man beachte die Möglichkeit einen komplexen Ausdruck über mehrere Zeilen zu verteilen und mit Kommentaren zu versehen.

TETRA ist in der Lage im Text eindeutig zu entscheiden welches der einander ähnlichen Token im Eingabetext vorliegt:

```
CguardParser::~CguardParser()
```

wird als DESTRUCTOR erkannt und

```
CguardParser::SetIgnoreScanner()
```

wird als DECLARATOR erkannt.

Die Erkennung arbeitet mit einem Algorithmus, der nach der größtmöglichen Abdeckung des Quelltextes mit dem regulären Ausdruck sucht. Auf diese Art ist es möglich Beschränkungen zu umgehen, die die von TETRA angewendete Topdown-Analyse sonst hat.

7.8.4 Regeln: block, outer_block

Die ***outer_block***-Produktion beschreibt den Funktionskörper.
Der Code einer C++-Funktion kann sich über viele Zeilen erstrecken und kompliziert verschachtelte Strukturen enthalten. Für den Zweck des guard-Projekts ist dies irrelevant. Wichtig ist nur die Einfügepositionen für CGuard (s.o.) zu finden.
Die Idee ist, dass sich in C++ aller Code zwischen geschweiften Klammern befindet und dass es zu jeder öffnenden eine schließende Klammer geben muss. Ein solcher Block ist daher wie folgt zu beschreiben:

```
"{"
  (
    block
  | SKIP
  )*
"}"
```

Innerhalb des geschweiften Klammerpaares befindet sich eine Abfolge von Code und Unter-Blöcken. Ein Funktionskörper unterscheidet sich in nichts von einem solchen Block außer, dass er der äußerste Block der Funktion ist. Die Position nach seiner öffnenden Klammer und vor seiner schließenden Klammer sind also genau die gesuchten Einfügepositionen. Hier werden die beiden Funktionen **print_at_enter** und **print_at_exit** aufgerufen, die dazu dienen die gewünschten guard-Funktionen in die Ausgabe zu schreiben.

```
"{"
{{ print_at_enter(xState); }}
(
  block   {{ out << xState.lp_copy(); }}
```

```
| SKIP    {{ out << xState.copy(); }}
)*
"}"
{{ print_at_exit(xState); }}
```

Achtung:
Die Projekt-Option zum Testen aller literaler Token muss abgeschaltet sein, da sonst möglicherweise einem literalen Token, das eigentlich nur in anderen Produktionen des Projekts gebraucht wird, hier der Vorzug vor einer SKIP-Erkennung gegeben wird, wenn es innerhalb eines Blocks vorkommt.

Die Funktionen *print_at_enter* und *print_at_exit* sind auf der Seite für die Klassenelemente definiert:

```
print_at_enter ::=
{{
  out << xState.copy() // {
      << "\n  CGuard G(\""
      << m_sScope
      << "\",\""
      << m_sName
      << "\");\n";
}}
```

Hier werden die Variablen *m_sScope* und *m_sName* zur Konstruktione eines "Wächters" benutzt.

```
print_at_exit ::=
{{
  out << "\n  G.stop();\n}";
  m_sName.clear();
  m_sScope.clear();
}}
```

Hier wird der Inhalt der Variablen *m_sScope* und *m_sName* glöscht.

7.8.5 Nachbesserung: '{' und '}' in Strings

Eine Komplikation ergibt sich, wenn geschweifte Klammern innerhalb eines in Anführungszeichen gesetzten Textes vorkommen, z.B. "{" oder "[{}]". Diese Zeichen würden dann fälschlich als öffnende oder schließende Klammer eines Code-Blocks interpretiert werden.
Die Alternativen innerhalb eines Blocks müssen daher um Strings erweitert werden:

```
STRING      "([^"]|\\")*"
```

Die **block**-Regel und entsprechend die **outer_block** Regel sehen dann so aus:

```
"{"
{{ print_at_enter(xState); }}
(
  block   {{ out << xState.lp_copy(); }}
| STRING  {{ out << xState.copy(); }}
| SKIP    {{ out << xState.copy(); }}
```

```
)*
"}"
{{ print_at_exit(xState); }}
```

Mit dieser Verbesserung lässt sich auch der Code parsen, den TETRA aus dem guard-Projekt selbst erzeugt.

7.9 Rechnung

Die wichtigsten Bedienungselemente von TETRA sollten bekannt sein.

In diesem Projekt wird auf Unterausdrücke regulärer Ausdrücke zugegriffen. Dies ist künftig nur in der Standard und der Professional Version des TextTransformers möglich.

Problemstellung:

Die Summe der Einzelposten einer Rechnung soll berechnet werden. Das Beispiel enthält nicht viel Neues. Es soll lediglich eine weitere Art der Anwendung des TextTransformers demonstrieren.

TETRA Programm:

Das Projekt befindet sich in dem Ordner:

\TextTransformer\Beispiele\Rechnung.

Dies einfache Projekt verwendet unmittelbar mit den Token verknüpfte Aktionen, in denen die Textdarstellungen der Beträge in die entsprechenden Zahlenwerte konvertiert werden.

7.9.1 Produktion

Die einzige Produktion dieses Projekts ist nach dem nun schon bekannten Schema aufgebaut, demnach der gesamte Text als Alternative von speziellen Ausdrücken: den Beträgen der Waren, und dem übrigen Text besteht, der unter Verwendung des SKIP-Symbols übersprungen wird:

```
{{double sum = 0.0;}}
(
 Betrag[sum]
|Betrag_[sum]
|SKIP
)+
{{out << sum << "\n";}}
```

Die Variable des Typs double wird an die Token Betrag bzw. Betrag_ übergeben, wo ihr der Wert des jeweils erkannten Betrages hinzuaddiert wird. Am Ende des Programms wird die Gesamtsumme ausgegeben.

7.9.2 Token

Als Betrag sollen Ausdrücke mit Vor- und Nachkommastellen erkannt werden, wie z.B.:

23,8
1,35
365,-

Betrag_

Für Beträge, deren Nachkommateil durch einen Strich ausgedrückt ist, wird ein gesondertes Token definiert, da ein Strich nicht ohne weiteres in eine Zahl zu konvertieren ist.

```
Name:       Betrag_
Parameter:  double& xSum
Text:       (\d+),\s?-
Aktion:     xSum += stod(xState.str(1));
```

Das Token ist als eine Folge von Ziffern definiert, auf die ein Komma und ein Strich folgt. Optional kann vor dem Strich ein Leerzeichen stehen.

Der Zahlenwert des Tokens wird allein durch die Vorkommastellen bestimmt. Durch die Klammerung von "\d+" kann auf die Vorkommastellen gesondert zugegriffen werden. Der Textabschnitt, der die Vorkommastellen repräsentiert wird durch xState.str(1) zurückgegeben. Diese Rückgabe wird unmittelbar an die Konvertierungsfunktion "stod" weitergereicht, die aus dem Text den entsprechenden Double-Wert macht. Der Wert wird zur bisherigen Betragssumme, die im Parameter xSum gespeichert ist, hinzuaddiert.

Betrag

Im allgemeinen enthalten Beträge einen bezifferten Nachkommaanteil. Diese werden durch das Token "Betrag" mit erfasst:

```
Name:       Betrag
Parameter:  double& xSum
Text:       (\d+),(\d\d?)
Aktion:     xSum += stod(xState.str(1) + "." + xState.str(2));
```

Der Nachkommateil besteht aus einer Ziffer, aus die optional eine zweite folgen kann. Die Funktion "stod" kann Betragsausdrücke, in denen ein Komma vorkommt nicht direkt konvertieren. "stod" verlangt einen Punkt als Trenner. Zur Umwandlung der Textdarstellung des Betrags in eine Zahl wird daher intermediär die Textdarstellung selbst in die Form konvertiert, die von "stod" verlangt wird.

7.10 XML

Die wichtigsten Bedienungselemente von TETRA sollten bekannt sein.

Problemstellung:

Ein Parser für XML-Dokumente soll erstellt werden.
XML (eXtensible Markup Language) ist ein weitverbreiteter Standard für eine fomale Sprache, die strukturierte Daten und ihre Formatierung beschreibt. Derart erlaubt XML den Austausch dieser Daten zwischen verschiedenen Anwendungen und über das WEB.

TETRA Programm:

Das Projekt befindet sich in dem Ordner:

\TextTransformer\Beispiele\XML

ISO_XML.ttp ist eine erste Fassung des Projekts, die sich eng an die Formulierung des XML-Standards hält.

XML.ttp ist die überarbeitete Fassung.

Anhand dieses Projekts wird die Konstruktion und Auswertung von Parse-Bäumen erläutert.

7.10.1 ISO-XML

Auf dieser Seite werden ein paar Hinweise gegeben, wie der TextTransformer XML-Parser aus den Standard-Spezifikationen für XML abgeleitet ist. Wer an diesen Einzelheiten nicht interessiert ist kann ohne weiteres mit der nächsten Seite fortfahren.

Der **XML-Standard** wird ausführlich beschrieben unter

http://www.xml.com/axml/testaxml.htm

Zur Spezifikation von XML wird dabei eine Extended Backus-Naur Form (EBNF) Notation verwendet, die ebenfalls standardisiert ist (siehe: http://www.cl.cam.ac.uk/~mgk25/iso-ebnf.html).

Die **standardisierte EBNF-Notation (ISO-EBNF)** ist glücklicherweise der des TextTransformers ähnlich, ist jedoch nicht für den praktischen Einsatz konzipiert. ISO-EBNF ist erstens sehr elementar, so dass es eine Unterscheidung von Token und Produktionen nicht gibt. Zweitens nimmt die Grammatik keine Rücksicht auf deterministische Erkennbarkeit, insbesondere ist sie nicht LL(1)-konform.
Zur Umformung der XML-Grammatik sind daher drei Schritte erforderlich:

1. ein Import-Projekt (quick and dirty) ähnlich dem Projekt zum Import von Coco-Parsern, wodurch die ISO-EBNF-XML-Regeln als TextTransformer-Produktionen importiert werden können.
2. alle Produktionen, die lediglich Zeichenfolgen beschreiben, werden in Token umgewandelt (siehe Anmerkungen unten).
3. LL(1)-Konflikte werden aufgelöst, ähnlich wie für den E-Mail-Adressen-Parser beschrieben.

Ein weiters Problem besteht darin, dass XML-Dokumente prinzipiell Unicode unterstützen, was der TextTransformer zur Zeit noch nicht leistet. (Die Option zur Erzeugung von Parser-Code auf Wide-Zeichen Basis ist in Arbeit:). Die ersten 128 Zeichen des ASCII-Codes und des UTF-8 codierter Unicode sind jedoch identisch, so dass der XML-Parser des TextTransformers, trotz entsprechender Vereinfachung der Token-Definitionen, die überwiegende Zahl von XML-Dokumenten lesen kann.

Weitere Anmerkungen zur Umformung von ISO-EBNF:

In ISO-EBNF gibt es einen Operator, zu dem es in der Tetra-Syntax kein Pendant gibt:

A - B passt auf jeden String der zu A passt aber nicht zu B

Einfach ist die Übersetzung dieses Operators dann, wenn es sich bei *A* und *B* um Zeichen oder Zeichenklassen handelt, da sich dann *A - B* zu einer Gesamtemenge akzeptierter Zeichen zusammenfassen lässt.
Handelt es sich bei *A* und *B* um Zeichenfolgen, so kann *B* entweder eine zulässige Alternative von *A - B* sein oder das Auftreten von *B* im Eingabetext bedeutet einen Syntaxfehler.

Beispiel:
```
ISO-EBNF:          CData ::= (Char* - (Char* ']]>' Char*))
                   CDSect ::= CDStart CData CDEnd
                   CDEnd ::= ']]>'
Tetra:             CData ::= ( Char )*
                   CDSect ::= CDStart CData CDEnd
                   CDEnd ::= ']]>'

ISO-EBNF:          PITarget ::= Name - (('X' | 'x') ('M' | 'm') ('L' | 'l'))
Tetra:             Name | XML EXIT
Tetra:             XML ::= [Xx][Mm][Ll]
```

Sehr häufig werden im ISO-EBNF Zeichenklassen als Reihen von alternativen Zeichen beschrieben. Soweit möglich sollten diese zu einer mit '[' und ']' geklammeren Menge zusammengefasst werden, da die das Scannen des Textes dadurch erheblich beschleunigt wird.

Beispiel:
```
ISO-EBNF:          S ::= (#x20 | #x9 | #xD | #xA)+
Tetra:             S ::= [ \t\r\n]+
```

Bei der Zeichenmenge *S* des gerade gegeben Beispiels kann noch ein Schritt weiter gegangen werden. Es ist genau die Menge der auszulassenden Zeichen. Diese wird in der ISO-EBNF nicht als solche spezifiziert. Vielmehr wird jede Stelle der Grammtik explizit angegeben, an denen *S* vorkommen kann oder muss. Dadurch wird zum einen die Grammatik unübersichtlich und zum anderen entstehen dadurch für LL(1)-Parser Konflikte.

Beispiel:
```
XMLDecl ::= '<?xml' VersionInfo EncodingDecl? SDDecl? S? '?>'
EncodingDecl ::= S 'encoding' ...
SDDecl ::= S 'standalone' Eq ...
```

Nach dem Erkennen von *VersionInfo* gibt es gleich drei Möglichkeiten, mit *S* fortzufahren. Wird *S* hingegen als Menge der auszulassender Zeichen definiert, ist die Regel LL(1). Auf *VersionInfo* folgt: 'encoding' | 'standalone' | '?>'

Wird *S* als Menge der auszulassenden Zeichen definiert, dann sollten allerdings Zeichenketten zwischen denen *S* nicht vorkommen kann zu Token zusammengefasst werden:

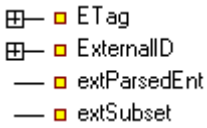
Beispiel:
```
ISO-EBNF:          EntityRef ::= "&" Name ";"
Tetra:             EntityRef ::= &{Name};
wobei {Name} ein Makro für die Zeichenmenge Name ist.
```

Als Problem bleiben Leerzeichen, die von der ISO-EBNF-Spezifikation an bestimmten Positionen gefordert werden. Es könnten in Tetra Token, auf die ein Leerzeichen folgen muss gesondert definiert werden. Damit wäre allerdings die Eleganz, die durch die Verwendung der auszulasenden Zeichen gewonnen wurde teilweise wieder verloren. Da der Beispielsparser nicht zur Verifikation der XML-Konformität, sondern dem Lesen und Verarbeiten von XML-Dokumenten dienen soll, muss man sich um diesen Punkt kein Kopfzerbrechen machen.

7.10.2 XML-Dokument

Die Startregel für den XML-Parser ist: ***document***. Nachdem die Startregel wie üblich mit

geparst wurde, erscheint in diesem Projekt im Syntaxbaum nicht vor allen Namen ein Kästchen zum Öffnen der Regelstruktur.

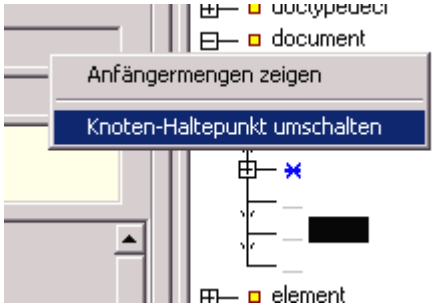


Das liegt daran, dass die XML-Grammatik aus zwei sich überschneidenden Teilen besteht:

- dem Teil für das eigentliche XML-Dokument und
- dem Teil für eine in eine andere Datei ausgelagerte DTD (document type definition), in der Tags und Attribute für ein XML-Dokument definiert werden.

Viele XML-Dokumente benötigen keine externe DTD und so soll es hier nur um den ersten Teil gehen. Die übrigen Regeln sind für interessierten Benutzer im Projekt verblieben.

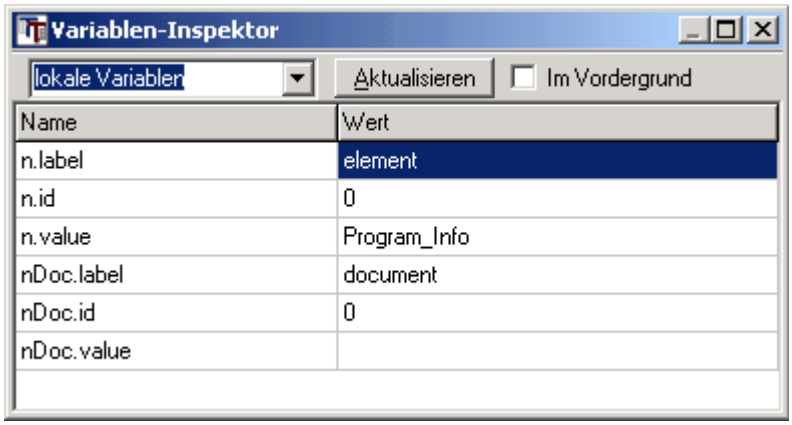
Sowohl die Grammatik als auch das Dokument selbst wirken auf den ersten Blick etwas verwirrend. Das TextTransformer-Projekt kann dazu dienen, beides deutlicher zu machen.
Die Struktur kann im Variablen-Inspektor dagestellt werden. Dazu wird ein Knotenhaltepunkt auf die semantische Aktion am Ende der Startregel gesetzt.



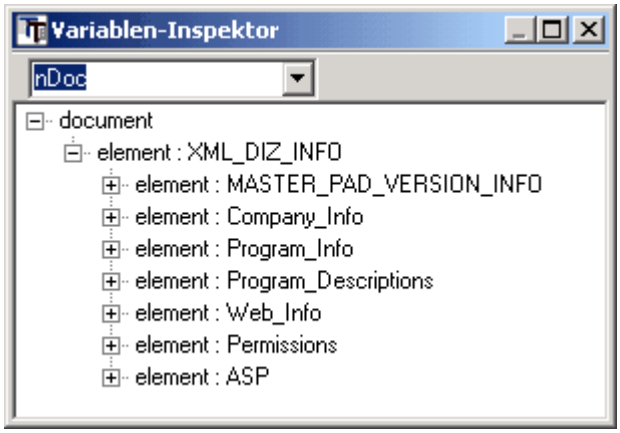
Nun wird das Programm im Debug-Modus bis zum Haltepunkt ausgeführt

und der Variablen-Inspektor wird aufgerufen.

Nun werden zunächst die lokalen Variablen zur Anzeige gewählt:



und auf der Werte-Seite wird ein Doppelklick auf eine der *nDoc*-Zeilen ausgeführt.



Das Ergebnis ist eine Baumdarstellung des XML-Dokuments.

7.10.3 Baumerzeugung

Das zuletzt angeklickte *nDoc* ist in der *document*-Produktion deklariert:

```
{{node nDoc("document");}}
```

node (Knoten) ist eine Struktur, die durch ein Label (hier: "document") und einen *str*-Wert charakterisiert ist. Außerdem haben Knoten die besondere Eigenschaft, dass sie sich baumartig miteinander verknüpfen lassen. Eine solche Verknüpfung mit anderen Knoten geschieht in den anderen Produktionen, die während des Parsens durchlaufen werden. Zunächst wird *nDoc* durch an die Produktion *element* übergeben:

```
element[ nDoc ]
```

Dort wird zunächst ein weiterer Knoten, nun mit dem Label: "element", deklariert:

```
{{node nElem("element"); }}
```

der wiederum an die *content*-Produktion weitergereicht wird:

```
content[nElem]
```

Nachdem die *content*-Produktion abgearbeitet ist, wird *nElem* als Kind.Knoten an xNode (das übergebene *nDoc*) angehängt.

```
xNode.addChildLast(nElem);
```

In gleicher Weise wurden mittlerweile beim Durchlaufen der *content*-Produktion andere Kind-Knoten an *nElem* angehängt. So entsteht der Baum, der dann im Variablen-Inspektor angezeigt werden kann.

Eine Alternative innerhalb der *content*-Produktion ist *CharData*. Das Token *CharData* bildet ein Blatt des Baumes. Hier wird ebenfalls ein node-Objekt erzeugt. Anders als die bisherigen Knoten erhält dieser jedoch einen Wert.

7.10.4 Baumauswertung

Wenn man die Struktur eines Dokuments zunächst in einen Baum umwandelt, statt sie unmittelbar in die gewünschte Form zu transformieren, so hat das den Vorteil, dass ein Zugriff auf die Daten nun mehrfach und in beliebiger Reihenfolge möglich ist. Die Ausgabe der Daten kann nun sehr systematisch organisiert werden und das Schreiben verschiedener Formate, z.B. Html und Rtf, kann durch einander ähnliche Prozeduren erfolgen.

In diesem Übungsbeispiel soll das XML-Dokument schlicht als einfacher Text ausgegeben werden.

Die Prozeduren zur Textausgabe der Baumdaten sind in einem gemeinsamen Container versammelt:

```
mstrfun           m_PrintText;
```

Bei dem Typ *mstrfun* handelt es sich um eine Funktionstabelle: *mstrfun* enthält Klassenfunktionen, die auf Knoten angewendet werden sollen, deren Label gleich dem Schlüssel ist, unter dem die Funktion in der Tabelle gespeichert ist.

Die Tabelle wird folgendermaßen initialisiert:

```
{{
  m_PrintText.add("", DontPrint);
  m_PrintText.add("document", DocText);
  m_PrintText.add("element", ElementText);
  m_PrintText.add("content", ContentText);
  m_PrintText.add("Attributes", AttributesText);
}}
```

D.h. für einen Knoten mit dem Label "document" soll die Funktion mit dem Namen "DocText" ausgeführt werden; für einen Knoten mit dem Label "element" soll die Funktion mit dem Namen "ElementText" ausgeführt werden ... In der ersten Initialisierungsanweisung ist eine Default-Funktion angegeben, die für alle Knoten ausgeführt werden soll, deren Label mit keinem der übrigen Schlüssel der Tabelle übereinstimmt.

Die Funktionen *DocText*, *ElementText*, *CharDataText* und *DontPrint* sind alle auf der Seite für Klassenelemente definiert. Die Funktion *DocText* ist für die Wurzel des Baumes aufzurufen:

```
{{
  node pos = xNode.firstChild();
  while(pos != node::npos)
  {
   m_PrintText.visit(pos);
   pos = pos.nextSibling();
  }
}}
```

Hier wird für alle Kind-Knoten des Knotens *m_PrintText.visit(xState, pos)* aufgerufen.
Die *visit*-Methode einer Funktionstabelle ist der Angelpunkt der gesamten Baumauswertung. Die Methode leitet das Knotenargument *pos* weiter an diejenige Funktion, die zum Label des Knotens passt. Man kann die *visit*-Funktion in diesem Beispiel als eine Abkürzung lesen für:

```
if(xNode.label() == "document")
{
 DocText(xNode);
}
else
if(xNode.label() == "element")
{
 ElementText(xNode);
}
else
if(xNode.label() == "CharData")
{
 CharDataText(xNode);
}
```

```
else
{
  DontPrint(xNode);
}
```

Die Funktion *ElementText* ist ebenso aufgebaut wie *DocText*, nur dass der Wert des Elements ausgegeben wird, bevor die Unterknoten besucht werden.

```
out << indent << xNode.value() << endl;
```

Auf diese Weise werden beginnend beim Wurzelknoten alle Unterknoten systematisch durchlaufen, bis schließlich die Blattknoten erreicht sind. Dort gibt die Funktion *CharDataText* ihren Wert aus.

7.10.5 Zeichen-Referenzen

Innerhalb eines XML-Elements

<text> ... </text>

dürfen die Zeichen:

< > " ' & $

nicht benutzt werden.Sie müssen daher entweder durch eine Namensentität oder eine dezimale Entität codiert werden:

Zeichen	Namensentität	Dezimale Entität
<	&lt	&#60
>	&gt	&#62
&	&amp	&#38
"	&quot	&#34
'	&apos	&#39

Zur Decodierung dieser Entitäten gibt es im XML-Projekt eine mstrstr-Klassenvariable: *m_EntityRefs* mit den Werten der ersten beiden Spalten der obigen Tabelle. In der *Reference*-Produktion werden mit Hilfe von *m_EntityRefs* die Namensentitäten in die Zeichen übersetzt.
Die entsprechenden dezimalen Entitäten werden in der Aktion zum Token:

```
CharRef ::= &#(\d+);|&#x([0-9a-fA-F]+);
```

behandelt, wo auch Codierungen für andere Sonderzeichen übersetzt werden. Denn auch Zeichen,

die nicht zu den ersten 128 Zeichen der ASCII-Tabelle gehören müssen u.U codiert werden. Ob und wie dies nötig ist hängt von encoding-Attribut in *XMLDecl* ab. Ein vollständiger XML-Parser müsste auf Tabellen für viele mögliche Zeichensätze zugreifen können. Hier wird zur demonstration vorausgesetzt, dass wir es mit dem Standard-Zeichensatz für Westeuropa, Lateinamerika (ISO-8859-1) zu tun haben. Dann können die Zeichen entsprechend der Numerierung der ANSI-Tabelle übersetzt werden.
Der reguläre Ausdruck *CharRef* erkennt entweder ein Zeichen in dezimaler Codierung und liefert die entsprechende Dezimalzahl als 1. Unterausdruck, oder er erkennt eine Hexadizmalzahl im 2. Unterausdruck.

```
{{
if(xState.length(1))
  return ctos(xState.itg(1));
 else
 if(xState.length(2))
    return ctos(hstoi(xState.str(2)));
  else
  {
    throw CTT_Error("unknown char reference");
    return str(); // formal return type
  }
}}
```

7.10.6 Kommentare und Verarbeitungsanweisungen

Es gibt noch eine Reihe von Schönheitsfehlern in ISO_XML.ttp, die nun im Übergang zu XML.ttp beseitigt werden sollen.

- Die überflüssigen Produktionen und Token werden entfernt
- Kryptische Abkürzungen, wie z.B."PI" werden durch aussagekräftigere Bezeichnungen ersetzt: ProcInstr (= Processing Instructions, Verarbeitungsanweisungen)
- Kommentare und Verarbeitungsanweisungen werden als Teil der auszulassenden Zeichen bzw. als Einschlüsse behandelt.

Der letzte Punkt dient didaktischen Zwecken. In anderen Grammatiken wäre er von größerem Nutzen als gerade bei XML, wo diese Einschlüsse nur an genau spezifizierten Stellen vorkommen dürfen.

Kommentare und Verarbeitungsanweisungen haben eine Sonderrolle: sie können an vielen Stellen im Dokument eingeschoben sein, ohne die Daten zu verändern, die durch das XML-Dokument zu einer Anwendung transportiert werden; sie enthalten zusätzliche Informationen.
Die Kommentare sind für den menschlichen Leser bestimmt und können von der Anwendung ignoriert werden. Der reguläre Ausdruck, der die Kommentare beschreibt kann mit den anderen auszulassenden Zeichen in einen Ausdruck zusammengefasst werden.

```
(\s|<!--([^-]|-+[^->]|->)*-+->)+
```

Verarbeitungsanweisungen enthalten Informationen für externe Anwendungen - z.B. können vollständige php-Skripte hier eingebettet sein - und können als Einschluss-Produktion gesetzt

werden.

Der neue Ausdruck und die Einschluss-Produktion können in den **globalen Projektoptionen** gesetzt werden. (*ProcInstr* muss dabei in den lokalen Optionen von sich selbst als Einschluss entfernt werden.) Dann toleriert der Parser allerdings auch XML-Dokumente in denen z.B. ein Kommentar innerhalb eines Tags vorkommt.
Soll ein derartiges Vorkommen einen Fehler erzeugen, müssen die Produktionen so verändert werden, dass sich ihre **lokalen Optionen** so modifizieren lassen, dass genau die erlaubten Vorkommen von Kommentaren und Verarbeitungsanweisungen erfasst werden. Ob es auszulassende Zeichen gibt oder, ob ein Einschluss folgt wird stets vor der Ermittlung des nächsten Tokens geprüft. Die lokalen Optionen einer Produktion hierfür werden also wirksam, sobald innerhalb der Produktion ein neues Token ermittelt wird. Da z.B. *content* mit *comment* beginnen kann, muss das Token das in der XML-Syntax als letztes vor *content* steht das erste Token einer Produktion sein, die auf Kommentare prüft. Deshalb wird die zusätzliche Produktion element_content, definiert (und analog die zusätzliche produktion doctypedecl_core). In den lokalen Optionen der folgenden Produktionen werden die Kommentare und Verarbeitungsanweisungen gesetzt:

```
content ::= ( element | CharData | "]]>" EXIT | Reference | CDSect )*
element_content ::=  content ETag
element_end ::=  "/>" | ">" element_content

doctypedecl_core ::= "[" ( markupdecl | PEReference )*

prolog ::= XMLDecl? doctypedecl?
```

Man beachte, dass auch in und nach der der *prolog*-Produktion Kommentare und Verarbeitungsanweisungen erkannt werden, da anders als bei anderen automatisch erzeugten Parsern, auch die Nachfolger von Produktions-Aufrufen explizit ermittelt werden.
Die element-Produktion ist nun ebenfalls etwas verändert. In ihr werden aber keine lokalen Optionen gesetzt.

```
element ::= "<" Name Attribute* element_end
```

7.10.7 Kundendaten übernehmen

Eine reale Anwendung zur Auswertung des XML-Baums ist die Überführung von Daten aus Kundenanfragen in eine Insert-Anweisung der Datenbank-Sprache SQL. Als Beispiel dienen die Texte Client1.txt - Client5.txt. Diese Texte sind keine vollständigen XML-Dokumente. Sie enthalten nach einem einleitenden Text, die Kundendaten in einer unvollständigen XML-Form.

Die Startregel ***Clients*** springt mit SKIP direkt zum Anfang des XML-Teils:

```
SKIP
"XML-Format:"
element[nDoc]
```

Dann wird die schon bekannte *element*-Produktion aufgerufen. Schließlich erfolgt die Konstruktion einer SQL-Anweisung in der Funktion: *PrintSQLInsert*. Mit

```
out << "INSERT INTO `tt_address` (`uid`, ...
```

werden die Tabelle und die Felder spezifiziert, die mit Werten gefüllt werden sollen. Die Werte werden dann aus dem Baum in der Reihenfolge ausgelesen, in der sie gebraucht werden. Eine Funktionstabelle ist in diesem einfachen Baum nicht nötig. Z.B.:

```
pos = xNode.findNextValue("EMAIL"); // email
out << pos.firstChild().value() << "\", \"";
```

So erhält man eine Insert-Anweisung, mit der die Werte in eine Datenbank eingefügt werden können. Z.B.:

```
INSERT INTO `tt_address` (`uid`, `pid`, `tstamp`, `hidden`, `name`, `title`,
`email`, `phone`, `mobile`, `www`, `address`, `company`, `city`, `zip`,
`country`, `image`, `fax`, `deleted`, `description`, `module_sys_dmail_category`,
`module_sys_dmail_html`) VALUES( "103", "43", "1087224349", "0", "  Santa
Clause", "", "sc@gift.org", "333 333", "", "", "", "", "North Pole", "",
"Greenland", "", "", "", "", "", "1" );
```

Mit einer N:1 Transformation im Transformations-Manager können die Insert-Anweisungen aller fünf Beispielsdateien in eine Textdatei geschrieben werden, so dass dann alle Datensätze auf einmal in die Datenbank eingefügt werden können.

7.11 Unit_Abhängigkeit

Die wichtigsten Bedienungselemente von TETRA sollten bekannt sein.

Problemstellung:

Häufig hängen Texte in verschiedenen Dateien voneinander ab. Ein typisches Beispiel hierfür sind Programmiersprachen, in denen diese Abhängigkeit durch sogenannte Include-Direktiven gekennzeichnet wird. In diesem Projekt soll am Beispiel der Programmiersprache Pascal demonstriert werden, wie man diese Direktiven nutzen kann, um auf die vorausgesetzten Texte zuzugreifen. Es soll eine Liste all derjenigen Dateien erzeugt werden, von der ein Pascal-Quelltext direkt und indirekt abhängt. Dazu ist es notwendig auch sämtliche vorausgesetzte Texte zu parsen.

TETRA Programm:

Das Projekt befindet sich in dem Ordner:

\TextTransformer\Beispiele\Unit_dependence

In diesem Projekt werden einige der Befehle zur Pfad- und Dateibehandlung verwendet und es wird gezeigt, wie Texte geladen werden, um sie dann in einem Unter-Parser zu verarbeiten.

7.11.1 Produktionen

In der Programmiersprache *Pascal* sind die vorausgesetzten Pascal-Units hinter dem Schlüsselwort *uses* aufgelistet. Z.B.:

```
uses
   Windows, Classes, SysUtils, Dialogs;
```

Einen Parser zu bauen, der nur diesen Textabschnitt behandelt ist sehr einfach:

```
unit ::=
    SKIP?
    (
        uses_clause
        SKIP
    )?

uses_clause ::=
    "uses"
    unit_name ("," unit_name )*
    ";"

unit_name ::=
    IDENT
```

Mit *SKIP* wird der Code übersprungen, bis das Schlüsselwort *uses* gefunden ist, dann wird die Liste der eingeschlossenen Units geparst und das Ende des Textes wird wieder übersprungen.

7.11.2 Container und Parameter

Zunächst müssen dem Programm die Verzeichnisse angegeben werden, in denen nach den Units zu suchen ist. Dazu wird auf der Seite für die Klassenelemente ein Vector-Container definiert

```
vstr  m_vIncludeDirs
```

Die Verzeichnisse muss der Benutzer angeben, bevor er das Programm startet. Z.B.:

```
m_vIncludeDirs.push_back("C:\\Programme\\Borland\\BDS\\4.0\\source\\Win32");
```

Die gesuchten Dateien müssen irgendwo gemerkt werden. Dazu wird ein *mstrstr* Container definiert:

```
mstrstr  m_mUnitPaths
```

In dieser Map kann zu jedem Namen einer Unit als Schlüssel der zugehörige Pfad als Wert gespeichert werden. Über den Schlüssel kann dann später leicht festgestellt werden, ob nach der Unit bereits gesucht wurde, falls sie in einer anderen Unit erneut aufgelistet ist.

```
if(!m_mUnitPaths.containsKey(sInclude)
...
```

Auch der Fall, dass der Pfad zu einer Unit nicht gefunden wird, muss gemerkt werden, um nicht erneut nach ihr zu suchen. Deshalb gibt es eine zweite Map:

```
mstrnode  m_mNotFoundUnits
```

Hier ist als Container-Typ mstrnode gewählt. Im node-Wert können gleich zwei Strings gespeichert werden, die von Interesse sein könnten: die Unit, in der die nicht gefundenen Datei aufgeführt ist und ein Level-Parameter, der Auskunft darüber gibt, wieviele Dateien geöffnet werden müssen, um von der ursprünglichen Quell-Datei zur ersteren Unit zu kommen.

Damit diese beiden Informationen erreichbar sind, müssen die oben genannten Produktionen mit

entsprechenden Parametern versehen werden.

```
    int  xiLevel, const  str&  xsLookedUpWhere
```

Die erste *unit*-Produktion wird dann mit den aktuellen Werten in der Startregel aufgerufen:

```
unit_dependence  ::=
    {{
    str  sWhereFound  = basename(SourceName());
    }}
    unit[0,  sWhereFound]
```

7.11.3 Dateien einschließen

Nun ist alles für den entscheidenden Schritt vorbereitet. Wenn der Name einer Unit gefunden wurde und nach der entsprechenden Datei noch nicht gesucht wurde, wird über die Such-Verzeichnisse iteriert, um den Pfad mit der Funktion find_file zu finden:

```
  vstr::cursor  cr  =  m_vIncludeDirs.getCursor();
  while(cr.gotoNext())
  {
    str  sPath;
    if(!is_directory(cr.value()))
       throw  CTT_Error(cr.value()  +  "  is  not  a  directory");
    if(find_file(cr.value(),  change_extension(sInclude,  ".pas"),  sPath))
    {
    ...
```

Dabei wird mit *change_extension* der die für Pascal-Dateien übliche pas-Dateierweiterung an den Namen angehängt. Vorsichtshalber wird zunächst mit *is_directory* noch geprüft, ob das Such-Verzeichnis auf Ihrem Computer vorhanden ist. Wenn Sie die Verzeichnis-Liste nicht modifiziert haben (s.o.) oder nicht können, weil Sie vermutlich kein Pascal installiert haben, so wird das Programm an dieser Stelle abgebrochen.

War die Suche erfolgreich, so enthält die Referenz-Variable *sPath* den gesuchten Pfad. Mittels load_file wird die Datei nun in den string *buf* geladen. Der Clou ist nun, dass die *unit*-Produktion wie eine normale Funktion aufgerufen werden kann, um den neuen Text zu parsen. Sie dient damit als "Unter-Parser":

```
    unit(buf,  ++xiLevel,  sInclude);
```

Während die *unit*-Produktion im Haupt-Parser nur zwei Parameter hat, bekommt sie für den Aufruf als Unter-Parser noch den zusätzlichen Text-Parameter an erster Stelle. In dem Unter-Parser wird nun genauso nach weiteren eingeschlossenen Units gesucht, wie zuvor im Haupt-Parser.

7.12 Java

Die wichtigsten Bedienungselemente von TETRA sollten bekannt sein.

Problemstellung:

Die Programmiersprache Java 1.4 soll geparst werden.

TETRA Programm:

Das Projekt ist die Adaption eines Coco/R-Projekt:

http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/Java/JavaGrammar.html

Das TextTransformer-Projekt *Java.ttp* befindet sich in dem Ordner:

\TextTransformer\Beispiele\Java

In diesem Projekt werden Vorausschau-Produktionen verwendet und einige spezielle Varianten der Vorausschau erklärt.
Die Anwendung des Baum-Assistenten und des Funktions-Tabellen-Assistenten werden Schritt für Schritt demonstriert, um zunächst den Code zur Erzeugung eines Parse-Baums zu generieren und dann ein Transformations-Programm (Kopier-Programm) zu erstellen.

7.12.1 Coco/R-Adaption

Die Coco/R-Adaption wurde großen Teils automatisch mittels eines TextTransformer-Programms vorgenommen, ähnlich wie es für die ältere Coco/R-Version beschrieben ist. Die IF-Konstrukte von Coco/R sind jedoch nicht äquivalent zu denen des TextTransformers und wegen ihres seltenen Einsatzes lohnt sich der Aufwand zur Entwicklung einer automatischen Übersetzung für sie nicht. Wegen dieses Mangels steht das Umwandlungsprogramm nicht zum Download zur Verfügung, auf Anfrage ist es aber kostenlos zu erhalten.

7.12.2 Einfache Vorausschau-Produktion

Der Java-Parser ist nicht LL(1)-konform. An mehren Stellen der Grammatik hängt die Entscheidung über die zu wählende Alternative von einer Vorausschau um mehr als ein Token ab. Das IF- und das WHILE-Konstrukt des TextTransformers erlaubt eine solche Vorausschau, wenn in der jeweiligen Bedingung eine Produktion zur Vorausschau aufgerufen wird.

Wird zum Beispiel in der *Statement*-Produktion ein Bezeichner als nächste Token erkannt, so ist zunächst nicht ausgemacht, ob dieser Bezeichner für ein Label steht oder für einen Ausdruck. Er steht für ein Label, wenn ihm ein Doppelpunkt folgt. Daher wird hier der Fortgang im Parser davon abhängig gemacht, ob die Produktion *isLabel*

```
isLabel ::= ident ":"
```

den anschließenden Text parsen kann oder nicht. `isLabel` kann den Text genau dann Parsen, wenn ein Doppelpunkt auf den Bezeichner folgt. Die Grammatik-Alternativen sind daher in die folgende IF-Konstruktion eingebunden

```
IF( isLabel() )
ident ":" Statement
ELSE
StatementExpression ";"
END
```

7.12.3 Negative Vorausschau

In der Produktion ArrayInitializer steht folgende WHILE-Schleife:

```
WHILE( commaAndNoRBrace() )
"," VariableInitializer
END
```

wobei

```
commaAndNoRBrace ::= "," ( "}" EXIT )?
```

Das bedeutet, die Schleife wird solange ausgeführt, wie ein Komma folgt, dem keine schließende geschweifte Klammer folgt. Folgt kein Komma, so kann commaAndNoRBrace den aktuellen Text nicht parsen. Folgt zwar ein Komma, diesem aber wiederum eine schließende geschweifte Klammer '}', so liefert die Vorausschau-Produktion commaAndNoRBrace ebenfalls false zurück. In diesem Fall wird dies durch EXIT erzwungen.

7.12.4 Komplexe Vorausschau

In der BlockStatement-Produktion wird die gleiche Produktion zur Vorausschau aufgerufen, die im Erfolgsfall ausgeführt werden soll:

```
IF( LocalVariableDeclaration() )
  LocalVariableDeclaration ";"
ELSE
(
    ClassOrInterfaceDeclaration
  | Statement
)
END
```

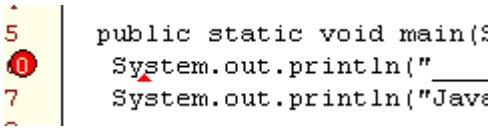
Hier ist **Vorsicht** geboten: je nachdem, ob in den Projekt-Optionen das Testen aller literaler Token aktiviert ist oder nicht, kann der Aufruf von *`LocalVariableDeclaration`* zur Vorausschau verschiedene Ergebnisse liefern. Folgt beispielsweise das Token *return* als eine der Alternativen von

Statement, so wird es als Bezeichner *ident* erkannt, wenn nicht alle literalen Token getestet werden, sondern nur die Anfängermenge von `LocalVariableDeclaration.` Damit wiederum würde "return true;" als eine lokale Variablendefinition erkannt werden, was sicherlich verkehrt ist.

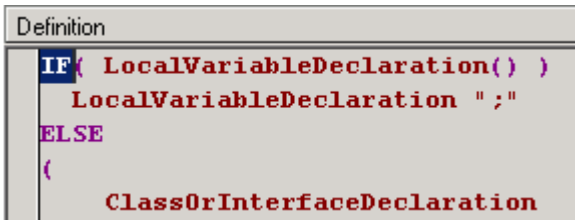
Anmerkung:
Beim Testen, ob `LocalVariableDeclaration` auf den aktuellen Text passt können indirekt weitere Vorausschau-Parser aufgerufen werden.

7.12.5 Vorausschau Debuggen

Was im einzelnen passiert, wenn eine Vorausschau getestet wird, kann man sehen, wenn man mit dem Debugger in die Vorausschau geht. Dazu kann zunächst ein Haltepunkt auf das Wort "System" in der sechsten Zeile des Beispieltextes gesetzt werden.

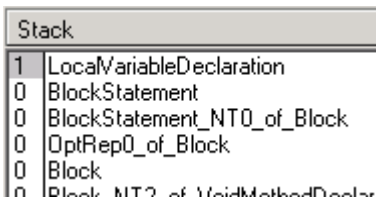


Dann kann das Projekt bis zu diesem Punkt ausgeführt werden. Nach drei weiteren Einzelschritten, wird in der Produktion "BlockStatement" das "IF" markiert.



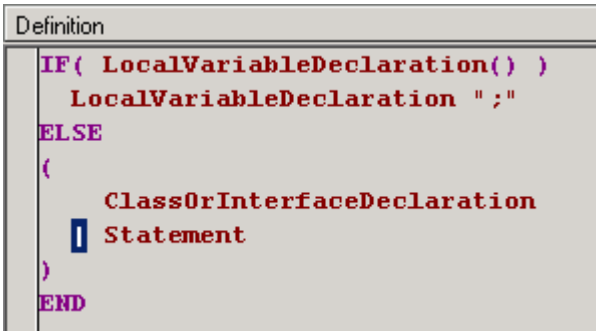
Nun kann in die Vorausschau gegangen werden. Nach Betätigung des Schalters wird zur Produktion "LocalVariableDeclaration" gewechselt und in dem kleinen Feld rechts neben dem Schalter eine '1' angezeigt.

Das bedeutet, dass der Debugger sich nun in der ersten Ebene einer Vorausschau befindet. Dies wird auch im Stack-Fenster durch eine grau hinterlegte '1' an der Spitze des Stacks vor dem Namen der Produktion gekennzeichnet.



Innerhalb der Vorausschau kann der Debugger nun genauso ausgeführt werden, wie man es schon

für den Hauptparser gewohnt ist. Hinter dem Wort "println" ist jedoch Schluss: es wird kein nachfolgendes Token erkannt. Die Vorausschau ist fehlgeschlagen und wird verlassen. Man befindet sich wieder in der Produktion "BlockStatement", nun aber in ELSE-Zweig der IF-Struktur.



Das Feld für die Anzeige der Ebene der Vorausschau ist leer, d.h. der Debugger zeigt wieder den Fortschritt im Hauptparser an.

7.12.6 Parse-Baum

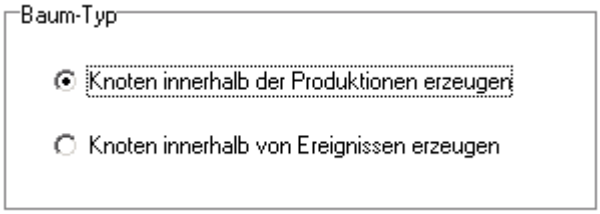
Der Java-Parser ist ein relativ umfangreiches Projekt und daraus ein vollständiges Transformations-Programm zu machen lässt eine Menge Arbeit erwarten. Hier können die Assistenten des TextTransformers eine große Hilfe leisten. Als Beispiel soll zunächst mit dem Baum-Assistenten der Code zur Erzeugung eines Parse-Baums in das Projekt eingefügt werden. Mit dem Funktionstabellen-Assistenten können dann Funktionen erstellt werden, mit denen der Parse-Baum ausgewertet werden kann.

Häufig ist es empfehlenswert, zunächst ein Transformations-Programm zu erstellen, dass die Quell-Dateien schlicht kopiert. Anhand eines solchen Programms lässt sich zum einen der Parser und der Parse-Baum gut testen: das Programm arbeitet korrekt, wenn die transformierten Dateien mit den Quelldateien identisch sind. Zum zweiten können an dem Kopierprogramm dann einfache Änderungen vorgenommen werden, deren Resultate ebenfalls leicht zu verifizieren sind.

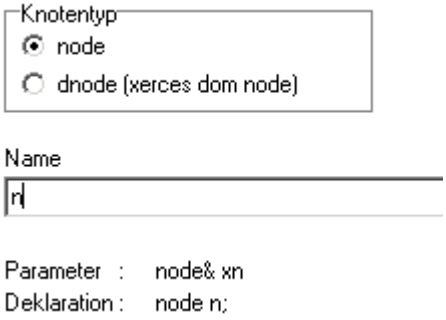
Wer die folgenden Schritte nicht im einzelnen nachvollziehen möchte, kann das fertige Ergebnis des Baum-Assistenten auch direkt laden:

...\TextTransformer\Beispiele\Java\JavaTree.ttp

Wenn man den *Baum-Assistenten* im Hilfe-Menü aufruft, so erscheint zunächst eine Auswahlmöglichkeit für die Art und Weise, wie der Baum erzeugt werden soll. Hier wird die Voreinstellung belassen.

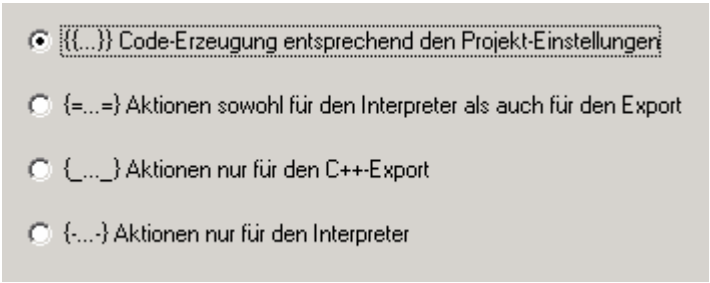


Als nächstes erscheint eine Auswahl für den Knotentyp und ein Eingabefeld für den Knoten-Namen. Hier wird *node* als Typ und "*n*" als Name gewählt:

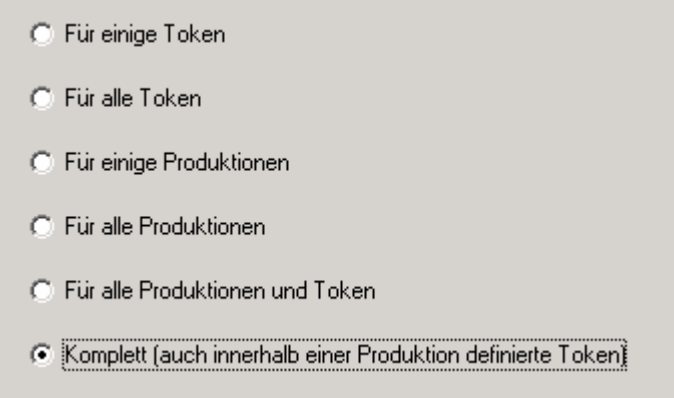


Unter dem Namensfeld wird bereits angezeigt, wie der Code für die Knoten-Parameter im Parameter-Feld und für die Knoten-Deklarationen im Text der Produktion für diesen Namen aussehen werden.

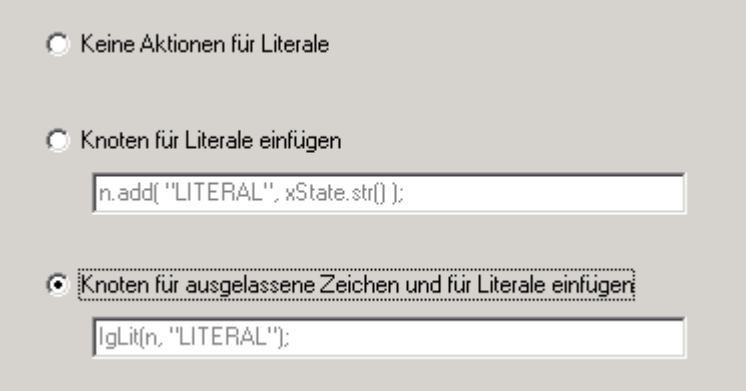
Auf der nächsten Seite des Assistenten belassen sie die Voreinstellung:



Wählen sie dann die Komplett-Option auf der nächsten Assistenten-Seite:



Auf der nächsten Seite des Assistenten stehen drei Optionen zur Behandlung der literalen Token zur Auswahl.



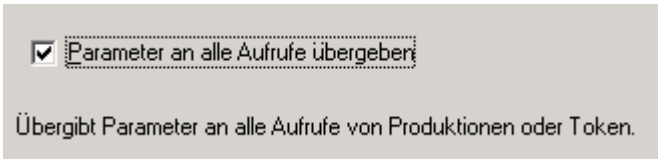
Für das Kopierprogramm muss der unterste Punkt gewählt werden.Dann wird nach jedem Vorkommen eines literalen Tokens im Projekt die semantische Aktion *IgLit* eingefügt, die dafür sorgt, dass dem Baum sowohl ein Knoten für die ausgelassenen Zeichen als auch ein Knoten für den erkannten Text hinzugefügt wird.

Die *IgLit* Funktion wird dann vom Assistenten auf der Element-Seite eingefügt. Sie sieht folgendermaßen aus:

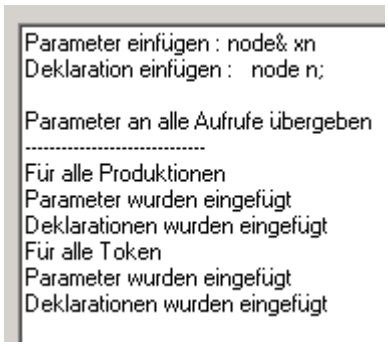
```
Name: IgLit  // Ignorierter Text und Literal
Parameter: node& xnNode, const str& xs
Text:
{{
  node n("IgLit");
  xnNode.addChildLast(n);
  n.add("IGNORED", xState.str(-1));
  n.add(xs, xState.str());
}}
```

Dem Baum-Knoten *xnNode* wird hier ein Unterknoten mit dem Label "Iglit" hinzugefügt, und dieser Unterknoten erhält die Knoten für den ausgelassenen Text und den vom Token erkannten Text.

Auf der nächsten Seite aktivieren sie bitte die Box: Parameter an alle Aufrufe übergeben.

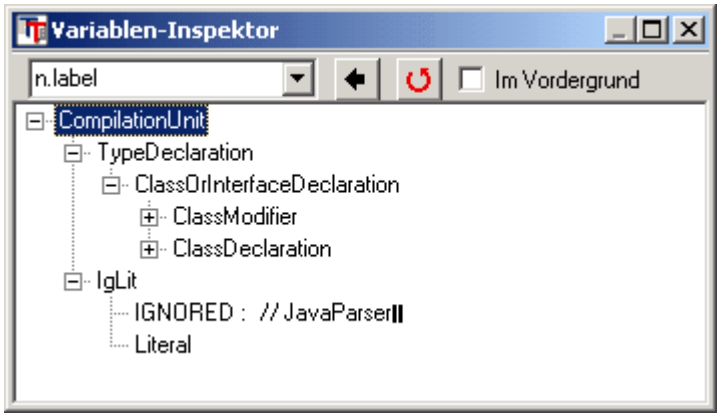


Dann können sie zur letzten Seite gehen und den Fertigstellen-Schalter anklicken.



Nachdem die Ergebnisse angezeigt wurden kann der Baum-Assistent mit dem Abbruch-Schalter geschlossen werden.

Der Baum, der nach Ausführung der Startregel *CompilationUnit* erzeugt wird kann im Variablen-Inspektor betrachtet werden, wie im XML-Beispiel beschrieben.



Anmerkung: An das Ende der Startregel *CompilationUnit* ist explizit das **EOF**-Symbol gesetzt. Für dieses Symbol ist vom Assistenten auch eine Aktion eingefügt worden:

```
EOF  {{IgLit(n, "Literal");}}
```

In dieser letzten Aktion wird auch der auszulassende Text der dem letzten Token der Java-Grammatik nachfolgt, in den Baum eingefügt:

der Kommentar `"// JavaParser"`

7.12.7 Funktions-Tabelle

Mit dem Funktions-Tabellen-Assistenten kann nun ein Gerüsst zur Auswertung des Parse-Baums erzeugt werden.

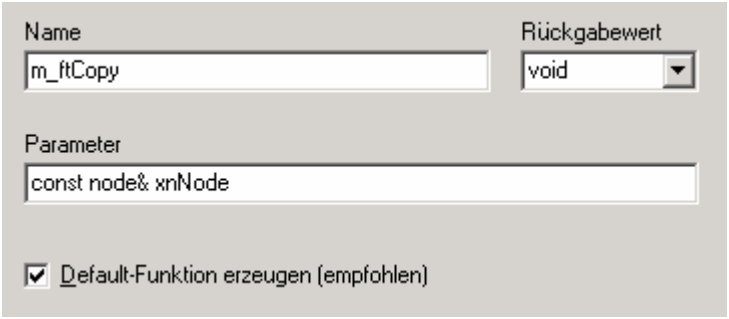
Wer die folgenden Schritte nicht im einzelnen nachvollziehen möchte, kann das fertige Ergebnis des Funktions-Tabellen-Assistenten auch direkt laden:

...\TextTransformer\Beispiele\Java\JavaCopy.ttp

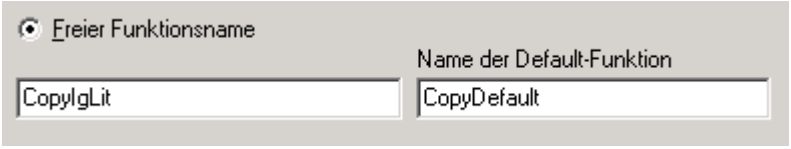
Wenn sie sich den Parse-Baum im Variablen-Inspektor angesehen haben, erkennen sie, dass der Baum aus Verzweigungen besteht deren Label die Namen von Produktionen sind. Für jedes Token gibt es einen Knoten mit dem Label: *IgLit*.
Nun soll eine Funktions-Tabelle erzeugt werden, die Funktionen zur Behandlung Knoten aller Label enthält. Für das Kopierprogramm sind vor allem die *IgLit*-Knoten wichtig, da in den beiden Unterknoten der gesamte Eingabetext vorhanden ist. Die Produktions-Knoten können alle gleich behandelt werden: sie dienen als Zwischen-Station bei der Iteration zu den IgLit-Knoten.
Es werden also insgesamt nur zwei Funktionen benötigt:

Default-Funktion zur Behandlung der Produktions-Knoten
IgLit-Funktion zur Behandlung der IgLit-Knoten

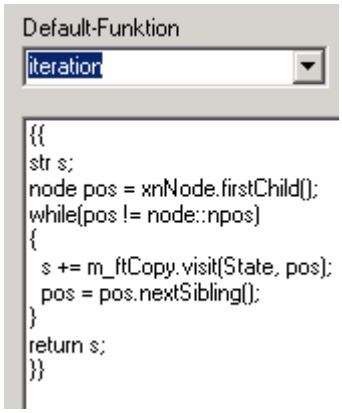
Wählen sie auf der Start-Seite des Funktions-Tabellen-Assistenten: Neue Tabelle erzeugen. Damit kommen sie auf die zweite Seite, deren Felder ausgefüllt werden sollten wie angezeigt:



Auf der nächsten Seite wählen sie : Funktion für ein einzelnes Label
Auf der nächsten Seite schreiben sie das Label: **IgLit**
Auf der nächsten Seite schreiben sie den Funktions-Namen: *Copy*IgLit und den Namen für die Default Funktion: *CopyDefault*.



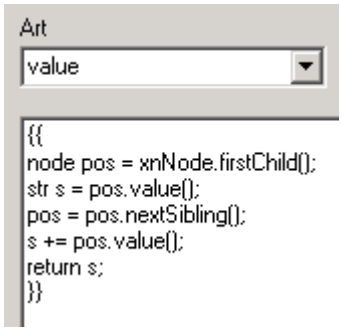
Als Default-Funktion wählen sie *iterate* auf der nächsten Seite.



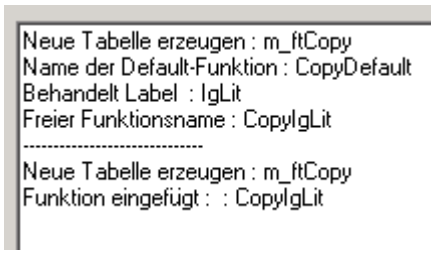
Diese Funktion dient der Iteration zu den *IgLit*-Knoten.
Auf der nächsten Seite wählen sie *value* und ändern den Text:

```
{{
node pos = xnNode.firstChild();
out << pos.value();
pos = pos.nextSibling();
out << pos.value();
}}
```

Diese Funktion gibt beide Text-Teile eines *IgLit*-Knotens aus.



Nun kommen sie auf die letzte Seite und klicken auf *Finish*.



Mit dem *Cancel*-Schalter können sie den Assistenten nun schließen.
Auf der Seite für die Klassenelemente können sie nun sehen, das eine Funktions-Tabelle und zwei Funktionen eingefügt wurden.

Das Kopier-Projekt ist fast fertig. Leider gibt es einen Schönheitsfehler. Der Baum-Assistent hat auf

der Token-Seite nur Code zum Hinzufügen von Knoten für den Token-Text erzeugt, den ausgelassenen Text aber nicht berücksichtigt.. Deshalb müssen die Aktionen auf der Token-Seite noch manuell geändert werden in:

```
{{IgLit(xn, "LITERAL");}}
```

Das letzte, was noch zu tun bleibt, ist der Aufruf zur Baumauswertung. Sie können

```
"/* Breakpoint */"
```

in der Aktion am Ende von *CompilationUnit* ändern zu:

```
m_ftCopy.visit(n);
```

Wenn sie das Projekt ausführen, wird der Java-Quelltext in das Zielfenster kopiert.

7.13 C-Typedef

Die wichtigsten Bedienungselemente von TETRA sollten bekannt sein.

Problemstellung:

Bisweilen können innerhalb eines Textes neue Namen als Abkürzung für komplexere Ausdrücke definiert werden. Solche Namen sollen während des Parsens als zusätzliche Token eingefügt werden.
Typisches Beispiel für solche Abkürzungen sind Typ-Definitionen in Programmiersprachen. In diesem TETRA-Projekt wird dies für die Sprache C demonstriert. Zur besseren Übersichtlichkeit wurden die Regeln von C stark reduziert. Die vollständige C-Grammatik wird demnächst unter

http://www.texttransformer.org

verfügbar sein.

TETRA Programm:

Das Projekt befindet sich in dem Ordner:

\TextTransformer\Beispiele\Typedef

In dem Beispiel wird die Verwendung von dynamischen Scannern mit Platzhalter-Token und Textbereichen demonstriert.

7.13.1 Typedef

Eine Typdefinition wird in C mit dem Schlüsselwort *typedef* eingeleitet. Z.B.:

```
typedef const char* cpchar;
```

Im Anschluss auf diese Definition kann *cpchar* anstelle von "const char*" im C-Code verwendet werden.
Die vereinfachte Produktion *type_definition* zum Parsen der Definition ist:

```
"typedef"
declaration_specifiers?   "*"*
ID
{{ AddToken(xState.str(),  "TYPE", ScopeStr()); }}
";"
```

Die zweite Zeile beinhaltet die Regel, der gemäß ein abzukürzender Ausdruck aufgebaut sein muss und mit *ID* in der dritten Zeile wird der Name für die Definition erkannt.
Mit der folgenden semantischen Aktion wird nun der gefundene Name als zusätzliche Alternative dem dynamischen Token "TYPE" hinzugefügt. *TYPE* wird auf der Tokenseite definiert als:

```
TYPE  ::=  {DYNAMIC}
```

TYPE wird in der *type_specifier* Produktion verwendet:

```
  "void"
| "char"
| "short"
| "int"
| "long"
| "float"
| "double"
| "signed"
| "unsigned"
| TYPE
```

So wird:

```
cpchar p;
```

korrekt als Deklaration einer Variablen p mit dem benutzer-definierten Typ *cpchar* erkannt.

7.13.2 Geltungsbereiche

Der dritte Parameter in dem eben erläuterten Aufruf

```
{{ AddToken(xState.str(), "TYPE", ScopeStr()); }}
```

wurde noch nicht erklärt. Mit diesem Parameter kann der Bereich festgelegt werden, innerhalb dessen, das zusätzliche Token erkannt wird. Wenn dieser optionale Parameter weggelassen wird, gilt die Definition für sämtlichen nachfolgenden Code. Wenn er aber angeben wird, so gilt die Definition nur so lange, wie ein Bereich mit dem entsprechenden Bereichsnamen definiert ist.
So ein Bereich wird gleich am Anfang der Startregel *translation_unit* definiert und an ihrem Ende wieder aufgelöst:

```
{{
PushScope("external");
}}
external_declaration*
{{
PopScope();
}}
```

Desgleichen geschieht im *compound_statement*:

```
"{"
{{PushScope(ScopeStr() + ".local" + itos(m_iLocalScope++));  }}
  ...
{{PopScope(); }}
"}"
```

Bereichsnamen werden mit einem Stack verwaltet. Der Befehl *PushScope* packt einen zusätzlichen Bereich auf den schon vorhandenen Stapel von Bereichen und *PopScope* entfernt den obersten Bereich. (Um eindeutige Namen für diese lokalen Geltungsbereiche erzeugen zu können, werden durchnummeriert.)
Solange wie der Bereich, für den ein dynamisches Token definiert ist sich noch im Stapel befindet,

solange wird das Token erkannt.
Im folgenden Beispiel wird innerhalb der ersten *compound_statement* der Typ *pchar* definiert.

```
if(xi > 0)
{
  typedef char* pchar;
   pchar p;
}
else
{
  pchar p;  // error
}
```

Im zweiten *compound_statement* gilt diese Definition jedoch nicht mehr, so dass ihre Verwendung zu einem Fehler führt. Damit ist genau der Mechanismus von Typdefinitionen in C nachgebildet.

7.14 TETRA-Produktionen

Die wichtigsten Bedienungselemente von TETRA sollten bekannt sein.

Im Verzeichnis

"\TextTransformer\Beispiele\Productions"

befindet sich das Projekt für die TETRA-Skriptsprache. Alle Aktionen sind aus den Produktionen entfernt.

7.15 TETRA-EditProds

Die wichtigsten Bedienungselemente von TETRA sollten bekannt sein.

Das Projekt im Verzeichnis

"\TextTransformer\Beispiele\EditProds"

ist ein Beispiel dafür, wie ein und derselbe Parser für verschiedene Zwecke verwendet werden kann, wenn er zunächst einen Parse-Baum erzeugt. Das Projekt *EditProds* ediert TETRA Produktionen in zwei nahezu gegensätzlichen Weisen:

1. Baumknoten können damit in Produktionen eingefügt werden
2. semantischen Aktionen können damit aus den Produktionen entfernt werden

7.16 TETRA-Interpreter

Die wichtigsten Bedienungselemente von TETRA sollten bekannt sein.

Im Verzeichnis

"\TextTransformer\Beispiele\Interpreter"

befindet sich das Projekt für den TETRA-Interpreters. Alle Aktionen sind aus den Produktionen entfernt.

Der Parser für den Interpreter basiert auf einer Grammatik die zu finden ist unter:

http://www.antlr.org/grammars/cpp

```
/*
 * PUBLIC DOMAIN PCCTS-BASED C++ GRAMMAR (cplusplus.g, stat.g, expr.g)
 *
 * Authors: Sumana Srinivasan, NeXT Inc.;           sumana_srinivasan@next.com
 *          Terence Parr, Parr Research Corporation; parrt@parr-research.com
 *          Russell Quong, Purdue University;       quong@ecn.purdue.edu
 *
 * VERSION 1.2
 *
 * SOFTWARE RIGHTS
 *
 * This file is a part of the ANTLR-based C++ grammar and is free
 * software.  We do not reserve any LEGAL rights to its use or
 * distribution, but you may NOT claim ownership or authorship of this
 * grammar or support code.  An individual or company may otherwise do
 * whatever they wish with the grammar distributed herewith including the
 * incorporation of the grammar or the output generated by ANTLR into
 * commerical software.  You may redistribute in source or binary form
 * without payment of royalties to us as long as this header remains
 * in all source distributions.
 *
 * We encourage users to develop parsers/tools using this grammar.
 * In return, we ask that credit is given to us for developing this
 * grammar.  By "credit", we mean that if you incorporate our grammar or
 * the generated code into one of your programs (commercial product,
 * research project, or otherwise) that you acknowledge this fact in the
 * documentation, research report, etc....  In addition, you should say nice
 * things about us at every opportunity.
 *
 * As long as these guidelines are kept, we expect to continue enhancing
 * this grammar.  Feel free to send us enhancements, fixes, bug reports,
 * suggestions, or general words of encouragement at parrt@parr-research.com.
 *
 * NeXT Computer Inc.
 * 900 Chesapeake Dr.
 * Redwood City, CA 94555
```

```
* 12/02/1994
*
* Restructured for public consumption by Terence Parr late February, 1995.
*
* DISCLAIMER: we make no guarantees that this grammar works, makes sense,
*             or can be used to do anything useful.
*/
```

7.17 TETRA-Import

Die wichtigsten Bedienungselemente von TETRA sollten bekannt sein.

Im Verzeichnis

"\TextTransformer\Beispiele\ImExport"

befindet sich das Projekt für denTETRA- Import von ASCII-Textdateien, die zuvor aus TETRA exportiert wurden. Alle Aktionen sind aus den Produktionen entfernt.

7.18 TETRA-Management

Im Verzeichnis

"\TextTransformer\Beispiele\Management"

befindet sich das Projekt zum Parsen eines Managements.

7.19 Cocor Import

Die wichtigsten Bedienungselemente von TETRA sollten bekannt sein.

Dies Beispiel ist für fortgeschrittene Benutzer des TextTransformers und Kenner des Compiler-Generators Coco gedacht.

Der TextTransformer wurde inspiriert von dem Compiler-Compiler Coco/R und ist ihm daher in vieler Hinsicht verwandt. Die Produktionen einer Coco/R Compilerbeschreibung lassen sich im wesentlichen ohne große Probleme in die Syntax des TextTransformers übersetzen. Das ist die Aufgabe dieses Projekts im Verzeichnis:

"\TextTransformer\Beispiele\CC2TT_17"

Als Quelltext-Beispiel dient das Skript **Cr_17.atg**
(im Original: Cr.atg. Die 17 ist hier zur Kennzeichnung der Versionsnummer von Coco/R gewählt. Das Java Project ist eine Adaption eines neueren Coco/R-Projekts.) In diesem Skript wird die Syntax der Coco/R Compilerbeschreibungssprache selbst definiert.

Um das Skript parsen zu können, muss der TextTransformer selbst die Definitionen des Skripts adaptieren. Er muss

die gleichen Zeichen auslassen
die gleichen Token erkennen
die Produktionen gleich parsen

7.19.1 Auszulassende Zeichen

Für Coco/R werden die auszulassenden Zeichen und Kommentare gesondert definiert. (Das Leerzeichen wird stets ausgelassen.) Zusätzlich gibt es Pragmas zur Steuerung des Compilers, die an beliebiger Stelle im Quelltext auftauchen dürfen. Im Skript Cr_17.atg steht dies in folgenden Zeilen:

```
IGNORE tab + eol + lf
PRAGMAS
  Options = "$" {letter}.
COMMENTS
  FROM "/*" TO "*/" NESTED
```

Im TextTransformer werden auszulassende Zeichen, Kommentare, und Pragmas zu einem regulären Ausdruck zusammengefasst und in den Projektoptionen gesetzt.

```
IGNORE:         [\r\n\t ]
PRAGMA =        \$[[:alpha:]]*
COMMENT =       /\*([^*]|\*+[^*/])*\*+/
```

ergibt ein neues IGNORE:

```
IGNORE =
        ([\r\n\t ]| \
        {PRAGMA}|{COMMENT})+
```

Anm: Verschachtelte Kommentare sind im TextTransformer nicht möglich.

7.19.2 Token

Eine automatische Übersetzung der Tokenspezifikationen von Coco/R in die regulären Ausdrücke des TextTransformers wäre prinzipiell machbar.
Dies soll hier aber nicht geschehen, da dies - insbesondere wegen der unterschiedlichen Art der Definition von Zeichenmengen - doch einigen Aufwand bedeuten würde. Die Tokendefinitionen machen zudem nur einen kleinen Teil eines Übersetzungsprojekts aus.
Um Coco/R Compilerbeschreibungen parsen zu können werden die Coco/R Tokendefinitionen deshalb hier direkt übersetzt.

Bei Coco/R werden zunächst die verwendeten Zeichenklassen definiert und diese bilden dann die Grundlage zur EBNF-Definition der Token. Im TextTransformer wäre diese Zwei-Schrittverfahren ebenfalls anwendbar, üblicherweise werden die Token jedoch zusammen mit ihren Zeichenklassen definiert. Hierbei kann auf eine Menge vordefinierter Zeichenklassen zurückgegriffen werden, die es für Coco/R nicht gibt.

Die entsprechenden Zeilen aus Cr_17.atg sind:

```
CHARACTERS
  letter   = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz_" .
  digit    = "0123456789" .
  cntl     = CHR(0)..CHR(31).
  tab      = CHR(9) .
  eol      = CHR(13).
  lf       = CHR(10) .
  back     = CHR(92) .
  noQuote1 = ANY - "" - cntl - back .
  noQuote2 = ANY - """ - cntl - back .
  graphic  = ANY - cntl .

TOKENS
  ident     = letter {letter | digit} .
  str    =   "" {noQuote1 | back graphic } ""
             | """ {noQuote2 | back graphic } """ .
  badstring =   "" {noQuote1 | back graphic } ( eol | lf )
             | """ {noQuote2 | back graphic } ( eol | lf ) .
  number    = digit {digit} .
```

Im TextTransformer können die Zeichenklassen folgendermaßen ausgedrückt werden:

```
noQuote1 =  [^\"[:cntrl:]\\]
noQuote2 =  [^'[:cntrl:]\\]
graphic  =  [^[:cntrl:]]
```

Damit ergibt sich für die Token:

```
IDENT        :: =    [[:alpha:]_]\w*
STRING       ::=     \"([^\"[:cntrl:]\\]|\\[^[:cntrl:]])*\"   \
                     |'([^'[:cntrl:]\\]|\\[^[:cntrl:]])*'
BADSTRING    ::=     \"([^\"[:cntrl:]\\]|\\[^[:cntrl:]])*(\r|\n)  \
                     |'([^'[:cntrl:]\\]|\\[^[:cntrl:]])*(\r|\n)
NUMBER       ::=     \d+
```

7.19.3 Produktionen

Coco/R benutzt zur Formulierung der Produktionen die *EBNF*-Syntax und einige spezielle Symbole. *EBNF* steht für "Enhanced Backus-Naur-Form". Diese Syntax leistet im wesentlichen das gleiche, wie die des TextTransformers, nur dass z.B. statt dem Klammerpaar für die Wiederholung "(...)*" das Klammerpaar "{...}" verwendet wird. Analog sind auch Vertauschungen anderer Klammerpaare vorzunehmen:

Coco/R	TETRA
[…]	(...)?
{...}	(...)*
(. ... .)	{_ ... _}
<…>	[…]

Das Coco/R-Symbol ANY kann zunächst durch SKIP ersetzt werden. Die Korrektheit dieser Ersetzung kann jedoch nicht automatisch garantiert werden. Sie muss für jeden Einzelfall geprüft werden.
Für die Coco/R-Symbole WEAK und SYNC gibt es kein Pendant im TextTransformer. Sie können bei der Transformierung weggelassen werden, da sie nichts über die Struktur des Eingabetextes besagen, sondern zur Fehlerbehandlung durch den Parser dienen.

7.19.4 Nachbearbeitung

Führt man mit dem CC2TT_17-Projekt die Transformation eines Coco/R-Skripts durch, so kann das Ergebnis als Datei mit der Erweiterung "ttr" abgespeichert und in ein neues Projekt auf der Produktionenseite importiert werden.
Da die Token und auszulassenden Zeichen nicht automatisch transformiert wurden, müssen sie "von Hand" dem Projekt hinzugefügt werden, so wie dies auf den beiden vorherigen Seiten für "Cr_17.atg" beschrieben wurde.
Die automatisch übersetzten Produktionen können noch Fehler aufweisen.

1.: Der Import ist daraufhin zu überprüfen, ob allen leeren Alternativen eine semantische Aktion

zugeordnet ist.

2.:Die einfache Ersetzung des ANY-Symbols durch das SKIP-Symbol wird häufig inkorrekt sein: Die entsprechenden Stellen müssen daher ebenfalls überprüft werden.

im Cr_17.atg-Projekt sind z.B. folgende Korrekturen vorzunehmen:

- (ANY)* kann schlicht durch SKIP ersetzt werden
- { ANY | badstring } muss ersetzt werden durch (SKIP | string | badstring)*, da sonst ein Text, in dem eine Anführung vorkommt, in den Teil vor und den Teil nach dem abschließenden Anführungszeichen zerlegt würde, d.h. in: SKIP badstring.

7.19.5 Semantische Aktionen

Die semantischen Aktionen werden im transformierten Text durch die Klammern {_ und _} als nicht interpretierbarer Code eingeschlossen, da kaum anzunehmen ist, dass sie vom TextTransformer interpretierbar sind.
Es besteht auch die Möglichkeit die semantischen Aktionen ganz **wegzulassen**. Hierzu deaktiviert man im Projekt CC2TT.ttp in den **lokalen Optionen** der Produktionen *Attribs* und *SemText* die Berücksichtigung des in doppelt geschweiften Klammern "{{...}}" gesetzten Codes für den Interpreter. Dadurch wird im Interpreter die Ausführung dieses Codes, der die semantischen Teile in die Ausgabe schreibt, unterdrückt. So bleibt nach der Transformation eine reine Parserbeschreibung für den Import in den TextTransformer. Entsprechend ist natürlich auch möglich den C++-Code für einen Parser generieren zu lassen, der die semantischen Teile des Ausgangstextes überspringt.

Anmerkung:

Für die *SemText*-Produktion sind die lokalen Optionen bereits aktiviert, um das **Testen aller literaler Token abzuschalten**. Dies ist notwendig, da *SemText* im Pronzip aufgebaut ist als:

```
SemText ::=  "(." SKIP ".)"
```

Wenn nun auf die öffnende Klammer "(." das literale Token "(" im Text folgt, so würde dies als nächstes Token erkannt werden, da literale Token stets den Token einer SKIP-Erkennung vorgezogen werden. Wenn hingegen der Scanner den Text nur auf diejenigen Token hin testet, die gemäß der aktuellen Regel erwartet werden, bilden alle übrigen im Projekt definierten Token für die Erkennung von *SemText* kein Problem.

TextTransformer

Teil VIII

8 Wie kann man ...

Hier werden Hinweise zu speziellen fortgeschrittenen Themen und ungewöhnlichen Konzepten gegeben.

8.1 Daten laden

Bisweilen werden in einem Projekt externe Daten benötigt. Einzelne Parameter können als Start-Parameter übergeben werden Es ist aber auch möglich größere Datenmengen aus einer externen Datei einzulesen.

Im folgenden Beispiel soll eine Vornamensliste dazu verwendet werden, um zu entscheiden, ob es sich bei einem mit einem Token *NAME* erkannten Namen um einen Vornamen handelt oder nicht. Das geht mit folgendem Code, wenn die Vornamen als Schlüssel in der map *m_mFirstNames* gespeichert sind.

```
NAME
{{
  if(m_mFirstNames.findKey(to_upper_copy(xState.str())))
     xsVorname = xState.str();
   else
     xsNachname = xState.str();
}}
```

Um die Vornamen in die map einzulesen, wird zunächst eine Datei mit den Namen in einen string *buf* geladen, der dann anschließend mit der Produktion *ReadFirstNames* geparst wird.

```
{{
str buf;

if(!load_file(buf,  "FirstNames.txt"))
  throw CTT_Error("\"FirstNames.txt\"  konnte  nicht  geladen  werden");

ReadFirstNames(buf);
}}
```

Die Produktion *ReadFirstNames* wird hier als Unter-Parser innerhalb der semantischen Aktion aufgerufen. Die Liste könnte so aussehen:

```
AARON
ACHIM
ADALBERT
ADALIA
ADAM
ADELBERT
ADELE
...
```

Sie ist sehr einfach zu parsen:

```
(
  SKIP     {{  m_mFirstNames[trim_right_copy(xState.str())]  =  "";  }}
  EOL
)*
```

8.2 Daten strukturieren

Im TextTransformer Interpreter können weder eigene Klassen noch Strukturen definiert werden. Wenn dies wirklich erforderlich ist, muss es im externen Code oder im nur exportierbaren Code geschehen. Für die meisten Zwecke, sind aber node/dnode als Strukturersatz völlig ausreichend. Die typischen Daten eines Angestellten lassen sich z.B. so zwischenspeichern, wenn die entsprechenden einzelnen Daten jeweils als strings vorhanden sind.

```
node  n(sVorname,  sNachname);
 n.setAttrib("Strasse",  sStreet);
 n.setAttrib("Wohnort",  sAddress);
 n.setAttrib("Geburtstag",  sBirthday);
 n.setAttrib("Gehalt",  sSalary);
```

Mehrere solche Datensätze können als Unterknoten in einem Baum verwaltet werden und Reihen von Knoten bzw. Bäumen können z.B. in einer *mstrnode* Klassenelement gesammelt werden.

8.3 Zusätzliche Zieldateien schreiben

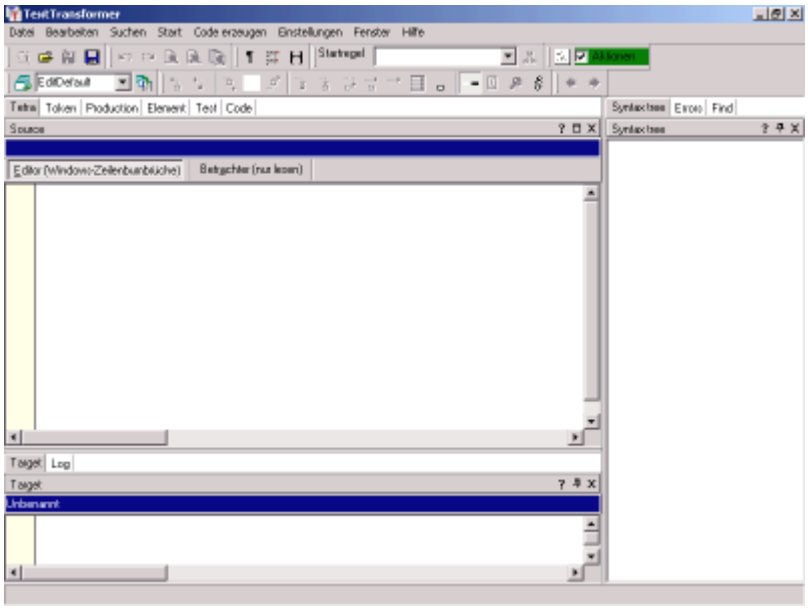
Grundsätzlich ist der TextTransformer an dem Modell orientiert, dass eine Quelldatei in eine Zieldatei umgewandelt wird. Im TextTransformer Interpreter können nicht mehrere Dateien gleichzeitig zum Schreiben geöffnet werden. Sind weitere Dateien zu schreiben, z.B. für log-Informationen etc., so ist es nur möglich die Ausgaben in eine andere als die ursprüngliche Datei umzuleiten und anschließend die Ausgabe auf die ursprüngliche Zieldatei zurückzusetzen.

```
{{
RedirectOutput(append_path(TargetRoot(),    xsPfad));
out << sLoginfo << endl;
ResetOutput();
}}
```

Teil IX

9 Bedienung

Nach dem Start des TextTransformers erscheint folgender Bildschirm:



Die Anordnung der Fenster wird durch das Layout "EditDefault" bestimmt. Sie können das Layout leicht mit der Maus an Ihren eigenen Bildschirm und Ihre eigenen Bedürfnisse anpassen.

Unterhalb des allgemeinen Menüs und der Werkzeugleiste befinden sich zwei Fensterblöcke mit jeweils einer Reihe von Tabulatoren.

Der linke Block umfasst Registerfenster in denen verschiedene Arbeiten ausgeführt werden können. In das große Fenster auf der aktiven Registerseite wird der zu verarbeitende Quelltext geladen. In dem darunter liegenden Ausgabefenster erscheint später das Ergebnis einer Transformation dieses Textes.

wie z.B. das Erstellen der TETRA-Regeln und ihre Ausführung. Der rechte Block enthält Fenster, die der Navigation innerhalb der TETRA-Skripte dienen.
.

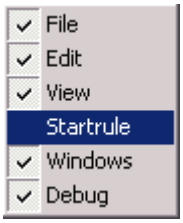
9.1 Werkzeugleiste

Gleich welche Fenster gerade aktiv sind, das Menü und die Werkzeugleiste und bleiben stets erreichbar. Die Inhalte des Menüs werden jedoch an die jeweiligen Fenster angepasst. D.h. einzelne Menüpunktes können hinzugefügt oder entfernt oder aktiviert bzw. deaktiviert werden.

Soweit die durch die Menüpunkte auslösbaren Aktionen nicht allgemeiner Natur sind, beziehen sie sich stets auf den aktuellen Arbeitsbereich. Zum Beispiel ist das Speichern des gesamten Projektes eine allgemeine Aktion, das Speichern des aktuellen Textes hingegen bezieht sich stets auf den jeweils gerade aktiven Editor.

Schaltergruppen lassen sich mit der Maus verschieben, oder ganz von der Werkzeugleiste abdocken und schließen.

Eine geschlossene Gruppe wird erneut angezeigt, wenn sie aus dem Menü ausgewählt wird, dass nach einem Klick mit der rechten Maustaste auf die Werkzeugleiste erscheint.



Die Positionen der Werkzeugleiste wird zusammen mit dem Fenster-Layout gespeichert.

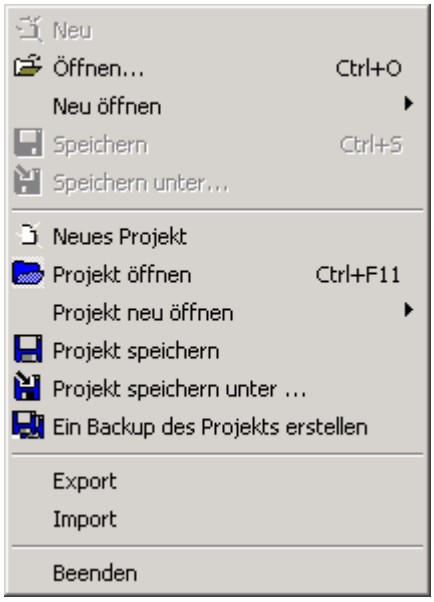
9.2 Hauptmenü

Die Haupt-Menüpunkte sind:

- Datei
- Bearbeiten
- Suchen
- Projekt
- Start
- Code erzeugen
- Einstellungen
- Fenster
- Hilfe

9.2.1 Menü: Datei

Das Dateimenu umfasst die Funktionen zum Öffnen und Speichern von Texten und Projekten:



Die obere Hälfte der Menüpunkte umfasst die Funktionen, die sich auf Texte beziehen, die untere Hälfte betrifft das Projekt.

Die Funktionen, die sich auf Texte beziehen betreffen immer das aktuelle Editorfeld! Ist z.B. das Quelltextfenster aktiv (d.h. der blinkende Text-Cursor befindet sich in diesem Fenster), so würde der Text dieses Fensters geöffnet oder gespeichert. Zum Speichern des Textes des Ausgabefensters muss das Ausgabefenster aktiv sein. Weitere Editoren gibt es in den Skriptverwaltungen oder auf der Editorseite.

Neu

Der aktuelle Text wird gelöscht

Öffnen...

Wird ein Quelltext geöffnet, so wird dieser im Betrachter angezeigt. In allen anderen Fällen wird der Text in das aktuelle Editorfenster geladen.

Unix-Zeilenumbrüche werden dabei automatisch zu Windows-Zeilenumbrüchen ergänzt

Eine Besonderheit ist die Auswahlbox ***Kodierung***:

Hier gibt es drei Möglichkeiten:

ANSI: der geöffnete Text wird als einfacher Windows-Text, d.h. als ANSI-codiert interpretiert. Jedes Byte der Datei wird im Editor als ein einzelnes Zeichen des ANSI-Zeichensatzes dargestellt.

UTF-8 (ANSI): der geöffnete Text wird als UTF-8 codiert interpretiert. TETRA kann nicht wirklich alle UTF8 kodierten Texte verarbeiten, sondern nur solche, die die sich in den ANSI-Satz transformieren lassen. Auch in diesem Modus werden Zeichen, die nicht zum ANSI-Zeichensatz gehören im Editor falsch dargestellt.

auto: XML-Dateien werden als UTF-8 codiert interpretiert, wenn UTF-8 in ihnen als Codierung angegeben ist. Alle anderen Texte werden als einfacher Windows-Text interpretiert.

Neu öffnen

Ein Text, der bereits einmal geladen war, wird erneut in das aktuelle Editorfenster geladen. Hierbei wird der Text stets im ANSI-Modus (s.o.) geöffnet.

Speichern

Der Text des aktuellen Editors wird gespeichert

Speichern unter...

Der Text des aktuellen Editors wird unter einem neuen Namen gespeichert.
Wie beim Öffnen eines Texts (s.o.) besteht auch hier die Möglichkeit verschiedene Codierungen zu wählen.

Neues Projekt

Ein neues Projekt kann entweder ohne Vorgaben angelegt werden oder es kann der Assistent zum Anlegen eines neuen Projekts aufgerufen werden.

Projekt öffnen

Ein vorhandenes Projekt wird über eine Dateiauswahlbox geöffnet. Es wird automatisch eine Kopie der bisherigen Version des Projekts unter dem alten Namen mit der Erweiterung "bak" angelegt.

Projekt neu öffnen

Ein Projekt, das bereits einmal in der TETRA-Umgebung bearbeitet wurde, wird erneut geladen. Es wird automatisch eine Kopie der bisherigen Version des Projekts unter dem alten Namen mit der Erweiterung "bak" angelegt.

Projekt speichern

Das aktuelle Projekt wird gespeichert

Projekt speichern unter ...

Das aktuelle Projekt wird unter einem neuen Namen gespeichert

Backup des Projekts erstellen

Im Projektverzeichnis wird ein Unterordner angelegt, in den die aktuelle Projektdatei kopiert wird und, falls vorhanden, die aktuellen Schablonen. Die Namen der Unterverzeichnisse werden zusammengesetzt aus "Backup" und einer dreistelligen Zahl. Die Zahl eines neuen Verzeichnisses ist stets um Eins größer als die größte Zahl in den Namen der anderen Backup-Verzeichnisse.

Import/Export

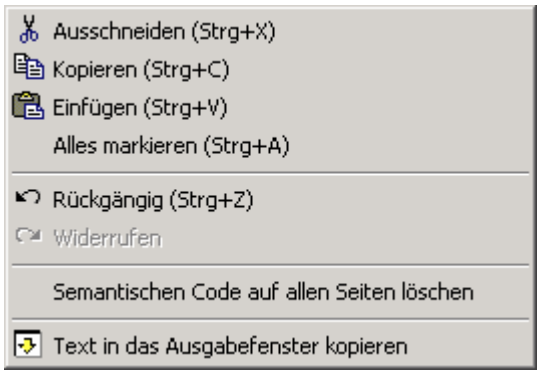
TextTransformer-Projekte werden normalerweise mit den oben genannten Funktionen in einem binären Format gespeichert und wieder gelesen. Mit den Funktionen zum Export bzw. Import können Projekte aber auch als in einer Text-Form gespeichert und gelesen werden.

Beenden

Das TETRA-Programm wird beendet.

9.2.2 Menü: Bearbeiten

Das Menü Bearbeiten umfasst im wesentlichen die üblichen Menüpunkte zum Ausschneiden, Kopieren etc., die sich in jedem Editor finden



Die Funktionen betreffen immer das aktuelle Editorfeld!

Rückgängig und Widerrufen

Die Funktionen: *Rückgängig* und *Widerrufen* arbeiten **zweistufig**:

- Befindet sich das Skript im Ediermodus, werden die einzelnen Edieraktionen rückgängig gemacht, oder rückgängig gemachte Aktionen erneut ausgeführt.

- Befindet sich das Skript nicht im Ediermodus wird der gesamte Zustand eines Skripts wiederhergestellt, den es hatte, bevor Änderungen an ihm akzeptiert wurden. Dies ist nur innerhalb einer Arbeitssitzung möglich.

Über den Menüpunkt

Semantischen Code auf allen Seiten löschen

lassen sich die mit den Token verbundenen Aktionen und der semantische Code in den Produktionen löschen. Außerdem werden alle Klassen-Elemente entfernt.

Mittels der Funktion

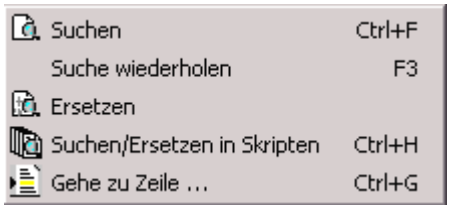
Text in das Ausgabefenster kopieren

oder den Schalter

lassen sich Textabschnitte, die untransformiert übernommen werden sollen in das Ausgabefenster kopieren. Im Eingabefenster muss der betreffende Abschnitt zunächst markiert werden. Nach Betätigung des Menüpunktes wird der Text an den Text des Ausgabefensters **angehängt**.

9.2.3 Menü: Suchen

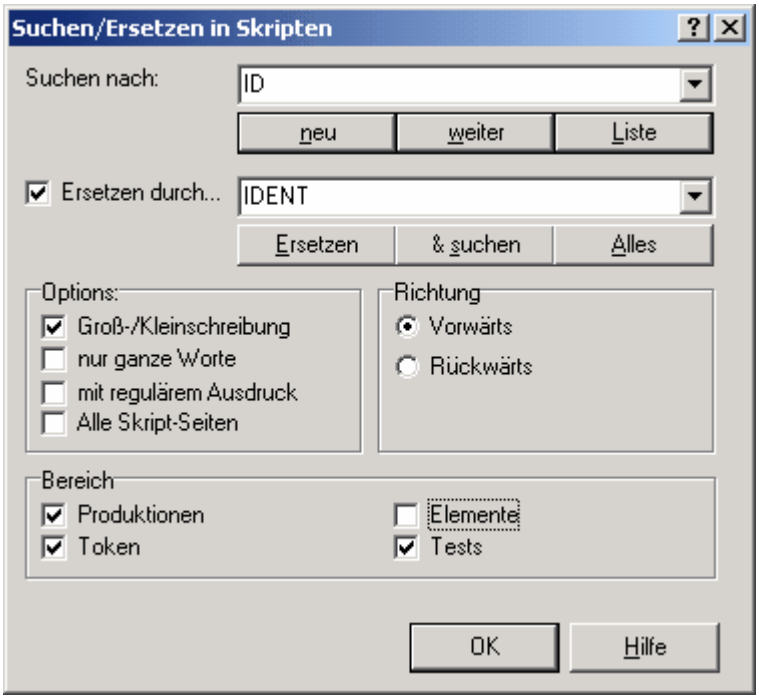
Das Menü *Suchen* umfasst Funktionen zum Suchen und Ersetzen von Texten im aktuellen Editor-Fenster.und in allen Skripten eines Projekts.



Die Funktionen
Suchen
Suche wiederholen
Ersetzen
Gehe zu Zeile...
betreffen immer das aktuelle Editorfeld!

Suchen und Ersetzen in Skripten

Die Arten der Skripte, in denen Ausdrücke gesucht oder ersetzt werden sollen, können im Dialog unten ausgewählt werden.Wenn mehrere Skript-Typen ausgewählt sind, erfolgt die Suche vorwärts in der Reihe Token, Produktionen, Elemente und Tests. Sämtliche Felder der Skripte werden durchsucht. Wenn nicht der Schalter *neu* betätigt wird, beginnt die Suche im aktuellen Skript an der aktuellen Position und wird dann in den Skripten mit den alphabetisch folgenden Namen fortgesetzt.



Wenn eine Liste der Suchergebnisse erstellt werden soll, so erscheint sie in einem speziellen Fenster: Search. Durch Anklicken der Einträge dort, kann zur entsprechenden Position navigiert werden.

9.2.4 Menü: Projekt

Das Menü *Projekt* wird nur angezeigt, wenn auch die Eingabemaske für ein Skript sichtbar ist. Hier sind alle Funktionen zum Verwalten und Kompilieren der Skripte zusammengefasst.

Neu
Übernehmen
Verwerfen
Löschen
Semantischen Code einklappen
Semantischen Code im Skript löschen
Semantischen Code in allen Skripten löschen
Skript kopieren : kopiert das aktuelle Skript in die Zwischenablage
Skript einfügen : fügt ein Skript aus der Zwischenablage ein
Kommentar : für jedes Skript kann ein Kommentar eingegeben werden
Lokale Einstellungen
Isoliert parsen
Zusammenhang parsen
Alle parsen
Import
Export

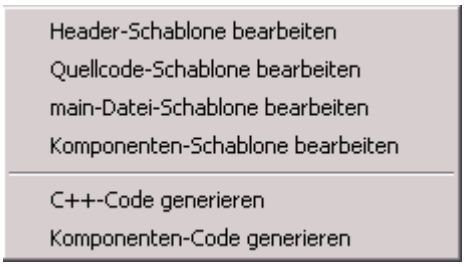
9.2.5 Menü: Start

Im Menü *Start* sind die Funktionen zum Debuggen und Ausführen von Projekten zusammengefasst.

Aktionen
Startregel parsen
Nächstes Token
Zurück zum letzten Knoten
Einzelner Knoten
Einzelner Knoten zurück
In die Vorausschau
Aus der Vorausschau zurück
Gesamte Verzweigung
Start
Ausführen
Dateigruppen transformieren
Zurücksetzen
Variable untersuchen
Text-Haltepunkt umschalten
Text-Haltepunkte löschen
Erkanntes/erwartetes Token markieren
Zur aktuellen Position
Letzte Meldung zeigen

9.2.6 Menü: Code erzeugen

Das Menü *Code erzeugen* enthält Funktionen die den Delphi-Entwickler bei der Benutzung der TextTransformer-Komponenten unterstützen, sowie die Funktionen der Professional Version des TextTransformers zur Bearbeitung der Code-Schablonen und für die Erzeugung von C++-Code.



C++-Code-Erzeugung

Header-Schablone bearbeiten öffnet einen Editor mit der Schablone für die Header-Datei der zu erzeugenden C++-Parser-Klasse.

Quellcode-Schablone bearbeiten öffnet einen Editor mit der Schablone für die Implementations-Datei der zu erzeugenden C++-Parser-Klasse.

main-Datei-Schablone bearbeiten öffnet einen Editor mit der Schablone für eine main-Datei oder eine andere Datei, in der der Parser aufgerufen wird.

C++-Code generieren stößt die Erzeugung des Codes für die C++-Parser-Klasse an.

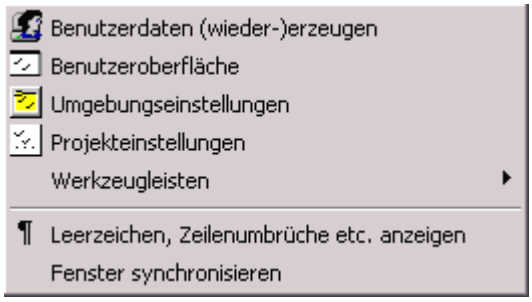
Delphi-Unterstützung

Komponenten-Schablone bearbeiten öffnet einen Editor mit der Schablone für die Delphi-Unterstützung

Komponenten-Code generieren erzeugt eine Pascal-Datei, zur Einbindung in Delphi-Anwendungen, die die TetraComponents verwenden

9.2.7 Menü: Einstellungen

Benutzerdaten erzeugen/wiederherstellen



Über das Menü *Einstellungen* sind drei Gruppen von Optionen einstellbar.

1. Einstellungen der Benutzeroberfläche
2. Umgebungseinstellungen
3. Projekteinstellungen

Für die einzelnen Produktionen gibt es weiterhin lokale Optionen.

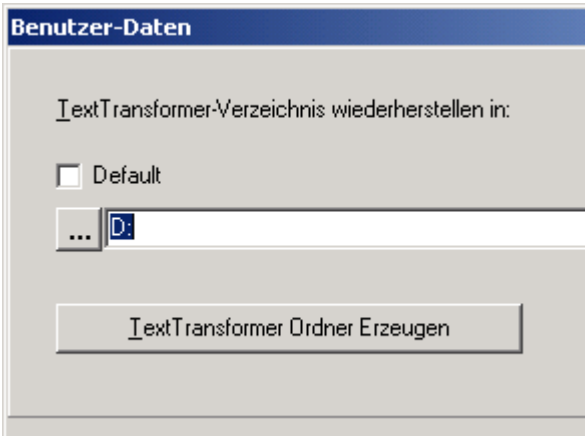
9.2.7.1 Benutzerdaten

Es gibt einige Einstellungen und Daten, die von individuellen Benutzern persönlich modifiziert werden können, ohne dabei die Entsprechenden Daten für andere Benutzer zu verändern. Zu diesen Daten gehören z.B. die Layouts, die Umgebungseinstellungen aber auch die Projektbeispiele.

Bei der Installation des TextTransformers wird eine Vorlage der Benutzerdaten in das Programmverzeichnis des TextTransformers geschrieben:

```
C:\Programme\TextTransformer\Data\TextTransformer
```

Beim ersten Programmstart wird der Dialog zur Erzeugung der Benutzerdaten automatisch aufgerufen um eine Kopie der Daten in einem Verzeichnis zu erstellen, auf das der Benutzer Zugriff haben muss, aber ansonsten frei wählen kann. Es ist möglich, das Benutzerverzeichnis später wiederherzustellen oder an anderer Stelle neu anzulegen.



Das Kopieren der Daten wird mit dem Schalter *TextTransformer-Ordner erzeugen* gestartet. Wenn es erfolgreich war wird der *Ok*-Schalter aktiviert und der *Abbrechen*-Schalter deaktiviert.

Ein existierendes *Settings*-Verzeichnis aus einer vorhergehenden Installation wird nicht überschrieben. Die Layouts der letzten Installation können über das Menü explizit in das Benutzerverzeichnis importieren.

Wenn der Ordner für die TextTransformer Daten erfolgreich erzeugt wurde, hat er folgenden Struktur:

```
TextTransformer
            \Backup
            \Beispiele
            \Frames
```

```
\Log
\Projects
\Settings
\Target
```

9.2.7.2 Einstellungen der Benutzeroberfläche

Die Einstellungen der Benutzeroberfläche sind im Hauptmenü als Unterpunkt zu den Einstellungen zu erreichen. Sie betreffen

die Ausführungsweise von Transformationen und
die Bearbeitung von Projekten
die Ansicht des Debuggers
die Layouts für den Edier- und den Debug-Modus

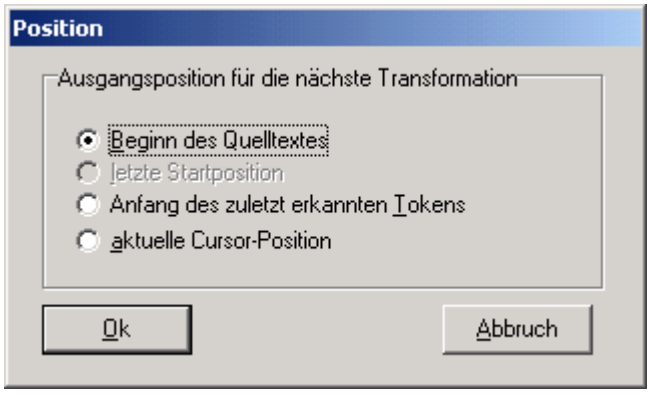
9.2.7.2.1 Transformation

1. STARTPOSITION:

Beim Starten einer Transformation kann

Immer am Anfang des Eingabetextes begonnen werden oder es wird, wenn nötig eine Auswahlbox möglicher Startpositionen angezeigt.

Ist ein Quelltext neu geladen, befindet sich der Textcursor zunächst in der linken oberen Ecke des Quelltextfensters und bestimmt damit die zunächst einzig mögliche Position an der mit einer Transformation des Textes begonnen werden kann. Sobald die Position des Cursors verändert wurde, sei es direkt über die Tastatur oder die Maus oder sei es, durch Transformation eines Teils des Textes, ergeben sich für den Neustart einer Transformation mehrere Möglichkeiten, von denen dann eine in folgender Box auszuwählen ist:



Nur die wirklich in Frage kommenden Möglichkeiten sind aktiviert. Kommt nur die erste der Möglichkeiten in Frage, wird die Box nicht angezeigt.

den **gesamten Text** verarbeiten
den **markierten Text** verarbeiten
den Text ab einer **vorherigen Startposition** erneut verarbeiten
den Text ab der Position des **zuletzt erkannten Tokens** verarbeiten
den Text ab der **aktuellen Cursor-Position** verarbeiten

Den gesamten Text verarbeiten

Unmittelbar nach dem Laden eines Quelltextes von der Festplatte befindet sich der Cursor am Anfang des Textes und beim Starten eines TETRA-Programms wird der gesamte Text verarbeitet.

Den markierten Text verarbeiten

Ist beim Starten der Transformierung ein Teil des Textes markiert, so wird nur dieser zur Verarbeitung ausgewählt.

Den Text ab einer vorherigen Startposition erneut verarbeiten

Befindet sich der Cursor beim Starten des TETRA-Programms nicht am Textanfang so besteht die Option erneut von der Position zu beginnen, an der die letzte Verarbeitung begann. Dies ist vor allem nützlich, um eine bestimmte Produktion an einem bestimmten Textabschnitt wiederholt zu testen.

Den Text ab der Position des zuletzt erkannten Tokens verarbeiten

Dies ist eine weitere Option des obigen Dialogs, die es ermöglicht eine Texttransformation interaktiv durchzuführen. Die Transformation kann so abschnittsweise durchgeführt werden, indem zwischenzeitlich der Resetschalter betätigt wird um eine neue Startregel auszuwählen mit der dann beim zuletzt erkannten Token neu angesetzt wird.

Den Text ab der aktuellen Cursor-Position verarbeiten

diese Option des obigen Dialogs dient ebenfalls der interaktiven Verarbeitung des Quelltextes. Hier kann nach Unterbrechung einer Transformation durch Reset, die Verarbeitung an der aktuellen Cursor-Position fortgesetzt werden. Wurde der Cursor nach dem letztmaligen Abbruch einer Transformation nicht mit der Maus oder den Curortasten an eine neue Position platziert, so ist die aktuelle Cursor-Position die Position am Ende des zuletzt erkannten Tokens.

2. AUSGABE:

Nachdem eine Transformation durchgeführt wurde, steht das Ergebnis der Transformation im Ausgabefenster. Wird unmittelbar nach der Transformation der Zurücksetzen-Schalter gedrückt, so erscheint eine Dialogbox, die das Löschen des Ausgabetextes anbietet.
Ist das Ausgabefenster bei Beginn einer Transformation nicht leer, so gibt es drei Optionen, wie TETRA sich verhalten soll:

1. Ausgabetext ohne Nachfrage stets löschen
2. Den neuen Ausgabetext an den alten anhängen
3. Einen Auswahldialog anzeigen

9.2.7.2.2 Edieren

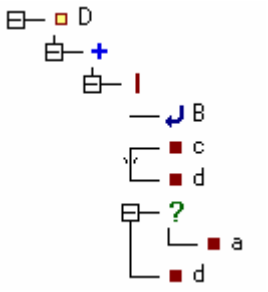
Änderungen an Skripten automatisch akzeptieren

Wenn sich ein Skript im Ediermodus befindet, d.h., wenn es neu ist oder gerade verändert wurde, so müssen die Änderungen übernommen werden, bevor andere Aktionen ausgeführt werden können, z.B. bevor zu einem anderen Skript gewechselt wird oder bevor das Projekt kompiliert wird. Wenn die Option *Änderungen an Skripten automatisch akzeptieren* aktiviert ist, so werden die Änderungen automatisch vor jeder solchen Aktion übernommen..

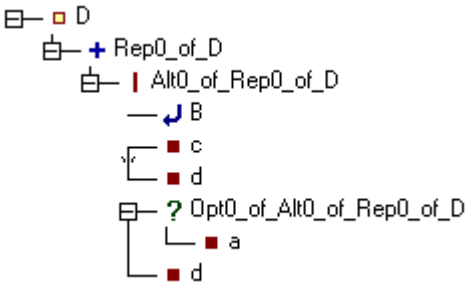
9.2.7.2.3 Ansicht

1. SYNTAXBAUM:

Hier kann eingestellt werden, ob nach dem kompilieren einer Produktion die Namen aller Knoten des Syntaxbaums angezeigt werden oder nur die Namen der Terminal-, SKIP- Nontteminalknoten (bzw. der Verzweigungen zu diesen). Im letzteren Fall ist die syntaktische Struktur deutlicher zu erkennen. Die Namen der Verzweigungen in Wiederholungen und Alternativen sind nur von Wichtigkeit, wenn sie mit dem produzierten C++-Code in Beziehung gesetzt werden sollen.



nur Non/Terminalnamen



alle Namen

2. MARKIERTES TOKEN:

Wenn eine Transformation schrittweise durchgeführt wird, ist die aktuelle Position im Eingabetext durch ein Token markiert. Je nach Einstellung ist diese das

zuletzt erkannte Token oder
das nächste erwartete Token

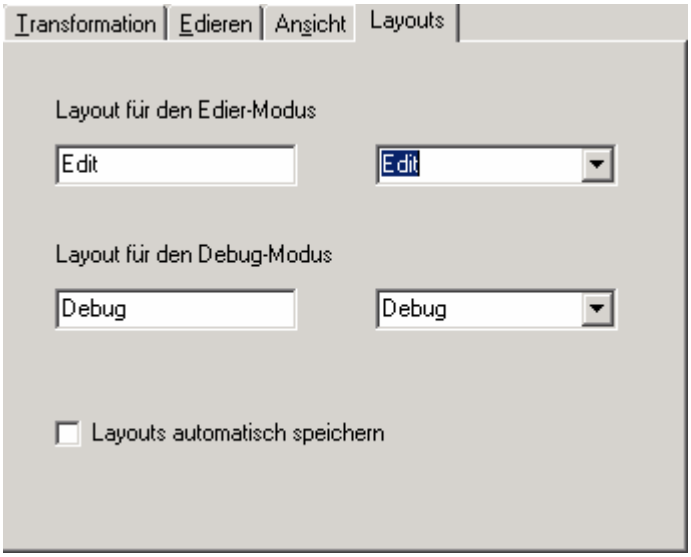
Hier kann ein Vorgabewert für das markierte Token gesetzt werden, der nach dem Start des TextTransformers aktiv ist. Dieser Wert kann während des Debuggens aber stets über den Schalter verändert werden:

 erkanntes Token anzeigen

 erwartetes Token anzeigen

9.2.7.2.4 Layouts

Wenn der TextTransformer zwischen Edier- und Debug-Modus wechselt wird automatisch auch das Layout gewechselt. Wenn die Einstellungen hierfür nicht geändert werden, werden jeweils die Default-Layouts verwendet. Auf der Layouts-Seite der Benutzer-Einstellungen ist es jedoch möglich, andere Layouts zu wählen.



Sämtliche Layouts, die im *Settings*-Ordner im DATEN-VERZEICHNIS gespeichert sind, sind in den Auswahlboxen in der auf der rechten Hälfte der Seite aufgelistet. Aus dieser Menge kann sowohl ein Layout für den Edier-Modus als auch eines für den Debug-Modus ausgewählt werden.

Wenn das Häkchen in der Check-Box zum automatischen Speichern der Layouts gesetzt ist,

werden die geänderte Layouts immer automatisch gepeichert, wenn der Programm-Modus geändert wird oder, wenn der TextTransformer beendet wird.

9.2.7.3 Umgebungseinstellungen

Die Umgebungseinstellungen zeigen auf der ersten Registerseite in einem Baum die Sektionen der Ini-Datei *tetra.ini*, die sich im gleichen Verzeichnis befindet wie das TETRA-Programm *tetra.exe* selbst.

CONFIG
EXTENSIONS
FRAMES
PATH

Auf der zweiten Registerseite gibt es die Möglichkeit, die Menge der Filter für die zu öffnenden Dateien zu verändern.

9.2.7.3.1 CONFIG

Hier läßt sich die Sprache der Benutzeroberfläche einstellen. Zur Auswahl stehen

German für Deutsch (default)
English für Englisch

9.2.7.3.2 EXTENSIONS

Hier lassen sich die Erweiterungen der Dateien bestimmen, die bei der Codeerzeugung generiert werden:

Schlüssel	Bedeutung	Default-Wert
Cpp_Header_Extension	Erweiterung erzeugter Header-	h
Cpp_Source_Extension	Erweiterung erzeugter Quellcode-Dateien	cpp
ComponentSupport_Extension	Erweiterung für Dateien zur Komponentenunterstützung	pas

9.2.7.3.3 FRAMES

Hier werden die Namen der Dateien angegeben, die per default das Gerüst für die Codegenerierung abgeben, bzw. zur Komponentenunterstützung dienen.

Schlüssel	Bedeutung	Default-Wert
Cpp_ParserHeader	Schablone für Header-Dateien	ttparser_h.frm

Cpp_ParserSource	Schablone für Quellcode-Dateien	ttparser_c.frm
ComponentSupport	Schablone für Komponentenunterstützung	enums_pas.frm

Anmerkung.: Für jedes Projekt können auch individuelle Schablonen erstellt werden, die im jeweiligen Projektverzeichnis zu speichern sind.

9.2.7.3.4 PATH

Die Installation des TextTransformer-Programm erfolgt in das PROGRAMM-VERZEICHNIS. Beim ersten Aufruf des Programm werden Sie aufgefordert für die Benutzerdaten ein DATEN-VERZEICHNIS zu erstellen.

Hier können Verzeichnispfade bestimmt werden

Frames, default "DATEN-VERZEICHNIS\Frames"

In diesem Verzeichnis sucht TETRA die Schablonen für die Codeerzeugung

Projects, default "DATEN-VERZEICHNIS\Projects"

Das Projekt-Verzeichnis ist das Stammverzeichnis für die verschiedenen Projekte, die mit TETRA erstellt werden.

Backup, default: "DATEN-VERZEICHNIS\Backup"

In das Backup-Verzeichnis werden die Dateien kopiert, die vor der Transformation von Dateigruppen gesichert werden sollen. Dies Verzeichnis kann interaktiv temporär verändert werden.

Target, default "DATEN-VERZEICHNIS\Target"

Das Target-Verzeichnis wird zunächst als Zielverzeichnis für die Transformation von Dateigruppen gesetzt, kann dann aber temporär interaktiv verändert werden.

9.2.7.3.5 Datei-Filter

Auf der zweiten Registerseite der Umgebungseinstellungen und im Dialog zur Transformation von Dateigruppen gibt es die Möglichkeit, die Menge der Dateimasken für die zu öffnenden Dateien zu verändern.

Die Liste führt alle Dateitypen auf, die zur Zeit zur Auswahl stehen, wenn entweder eine Datei in das Eingabefenster geladen werden soll, oder wenn eine Gruppe von Dateien für eine Transformation ausgewählt werden soll. Jeder Eintrag in dieser Liste kann ausgewählt werden um ihn zu löschen oder zu verändern.

Um einen Filter hinzuzufügen, wird zunächst der Schalter *Neu* betätigt. Nun kann die Beschreibung und der eigentliche Filter in die ensprechenden Felder geschrieben werden. Es ist möglich, mehrere Masken in einem einzigen Filter anzugeben. Hierzu sind die einzelnen Masken durch Semikolons zu trennen. Z.B.:

Beschreibung: Pascal files
Filter: *.PAS;*.DPK;*.DPR

Wenn eine Maske hinzugefügt, entfernt oder verändert wurde und der Dialog mit *Ok* beendet wird, wird die gesamte Liste gespeichert, so dass sie auch bei der nächsten Benutzung des TextTransformers zur Verfügung steht.

9.2.7.4 Projekteinstellungen

Die Projekteinstellungen gelten jeweils für das aktuell geladene Projekt. Sie werden automatisch zusammen mit diesem gespeichert, können aber auch individuell in ASCII-Form gesichert und geladen werden. Hierzu gibt es in der Dialogbox der Projektoptionen ein eigenes Menü.
Dateien für Projekteinstellungen erhalten die Erweiterung: **tto.**

Die Optionen sind auf mehrere Registerseiten verteilt:

Namen und Verzeichnisse
Parser/Scanner
Einschlüsse
Kodierung
Warnungen/Fehler
Code-Erzeugung

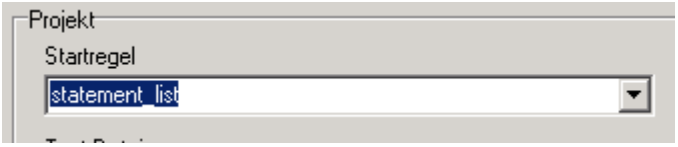
9.2.7.4.1 Namen und Verzeichnisse

Auf der ersten Seite der Projekteinstellungen können Namen und Verzeichnisse ausgewählt werden. Verzeichnisse werden relativ zum Projekt bestimmt.

Startregel
Test-Datei
Schablonenpfad

9.2.7.4.1.1 Startregel

Aus der Liste der Produktionen kann eine Startregel ausgewählt werden, die beim Öffnen des Projekts gesetzt werden soll.



Wenn hier keine Startregel ausgewählt wurde, so wird beim Öffnen des Projekts diejenige Produktion als Startregel gesetzt, deren Namen mit dem Namen des Projektes identisch ist. Gibt es keine solche Regel, wird keine Startregel gesetzt.

9.2.7.4.1.2 Test-Datei

Es ist möglich eine Datei auszuwählen, die beim Öffnen eines Projektes in das Quelltext-Fenster geladen wird. Dies ist vor allem wünschenswert, solange sich das Projekt noch in der Entwicklung befindet.

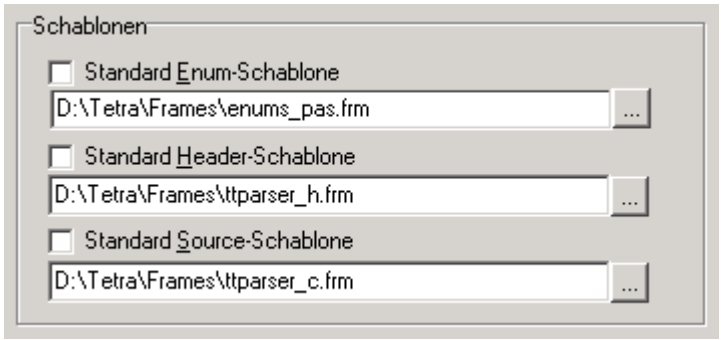
9.2.7.4.1.3 Präprozessor

Quelldateien können bereits beim Laden durch einen Präprozessor vorverarbeitet werden. Der Präprozessor ist eine weitere Instanz des TETRA-Interpreters die ein TETRA-Projekt ausführt. Dies Projekt kann hier gesetzt werden.

Im Präprozessor-Projekt kann ebenfalls ein Präprozessor-Projekt gesetzt sein. So kann eine ganze Kette von Vorverarbeitungen stattfinden.

9.2.7.4.1.4 Schablonenpfad

Hier wird der Ort bestimmt, an dem die Schablonendateien gespeichert werden, die dem vom TextTransformer erzeugten Code zugrunde liegen.



Die Default-Schablonen befinden sich im FRAMES-Verzeichnis.

enums_pas.frm ist die Standard-Schablone für die Komponentenunterstützung.

ttparser_h.frm ist die Standard-Schablone für die C++-Headerdatei
ttparser_c.frm ist die Standard-Schablone für denC++-Code

Diese Standardschablonen können für das jeweilige Projekt modifiziert werden. Sinnvoll ist es, die modifizierten Schablonen zusammen mit den übrigen Projektdateien in ein gemeinsames Verzeichnis zu speichern.

9.2.7.4.2 Parser/Scanner

In den Projekt-Optionen gibt es folgende Einstellungen für Parser und Scanner.

Auszulassende Zeichen
Groß-/Kleinschreibung
Wortgrenzen
Parameter und {{...}}
Globaler Scanner
Keine Fehlschlagsalternative zu SKIP
Keine Fehlschlagsalternative zu ANY

9.2.7.4.2.1 Auszulassende Zeichen

Beim Zerlegen von Texten in Symbole können überflüssige Zeichen übersprungen werden. Dies kann die Formulierung der Regeln stark vereinfachen. Zeichen die für die Zerlegung des Textes bedeutungslos sind können hier in den Projektoptionen global für alle Regeln ausgewählt werden. Die auszulassenden Zeichen haben keine Auswirkungen auf die regulären Ausdrücke selbst, sondern lediglich auf die Textpostionen von denen aus nach dem nächsten Token gesucht wird.

Beispiel:

Eine Regel für die Summe zweier Terme wäre intuitiv einfach zu formulieren als:

```
Summe = Term "+" Term
```

Durch diese Regel soll aber nicht nur ein Rechenausdruck wie

```
"23+4"
```

erkannt werden sondern auch

```
"23 + 4" und " 23 +  4" etc.
```

Die Leerzeichen zwischen den Zahlen und dem Plus-Operator sind irrelevant und können übersprungen werden. Ist das Leerzeichen nicht in den Optionen als auszulassendes Zeichen eingestellt, so müssen ein zusätzliche Token für die Lücke zwischen den Termen und dem Operator definiert werden. Möglich wäre

Space = "[\n\r\t]*" (umfasst auch Zeilenumbrüche und Tabulatoren).

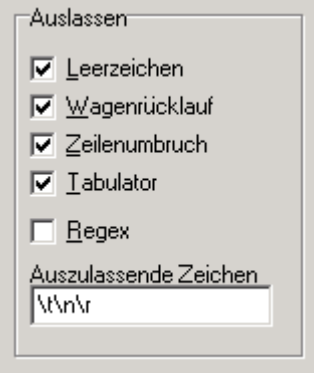
Statt obiger Regel wäre zu formulieren:

Summe = Term Space "+" Space Term

Je nach dem, ob die Checkbox *Regex* aktiviert ist oder nicht, werden die auszulassenden Zeichen durch eine Zeichenliste oder durch einen regulären Ausdruck definiert.

auszulassenden Zeichen mittels einer Zeichenliste definieren

Leerzeichen, Zeilenumbrüche und Tabulatoren dienen in sehr vielen Texten zur Trennung der Token und können bei der Analyse übersprungen werden. Sie sind daher standardmäßig als auszulassende Zeichen eingestellt und können bequem mittels der entsprechenden Checkboxen der Liste hinzugefügt oder entfernt werden.



Weitere Zeichen können direkt der Liste hinzugefügt werden.

auszulassenden Zeichen mittels eines regulären Ausdrucks definieren

Statt eine Liste der auszulassenden Zeichen zu definieren, kann auch ein regulärer Ausdruck zur Bestimmung des auszulassenden Textes verwendet werden. Hierzu muss das **Kästchen Regex** aktiviert werden. Der Text des Editfeldes wird nun als regulärer Ausdruck interpretiert. Beispielsweise könnte hier "\s*" eingegeben werden,. Dann würden alle Zeichen der Menge \s ausgelassen werden. Das ergäbe in etwa das gleiche Resultat wie die Zeichenliste die erhalten wird, wenn sämtlichen Checkboxen markiert sind. Ein sinnvolleres Beispiel für die Verwendung eines regulären Ausdrucks an dieser Stelle wäre:

(\s|//[^\r\n]*)*

Dieser Ausdruck veranlasst nicht nur das Überspringen der Leerzeichen sondern auch von Zeilenkommentaren.

Statt eines expliziten regulären Ausdrucks kann auch der **Name eines bereits definierten Tokens** in das Editfeld eingegeben werden, das dann zur Bestimmung der auszulassenden Textabschnitte verwendet wird.. Bei aktivierter Regex-Checkbox wird der Text im Editfeld als Tokennamen interpretiert, wenn er nur aus Literalen besteht.

Anm: Einem regulären Ausdruck, der die auszulassenden Zeichen definiert wird automatisch nach vorheriger Klammerung der Anker "\\A" vorangestellt. Damit wird gewährleistet, dass die übersprungenen Textabschnitte stets an der aktuellen Position beginnen.

Anm.: Während es bei der Verwendung von Zeichenlisten möglich ist mit xState.str(-1) auf die ausgelassenen Zeichen zuzugreifen, die einem SKIP-Knoten folgen, ist dies bei Verwendung eines regulären Ausdrucks nicht möglich.

9.2.7.4.2.2 Groß-/Kleinschreibung

Bei deaktivierter *Groß-/Kleinschreibung*-Option sind Token, die sich nur in ihrer Groß- bzw. Kleinschreibung unterscheiden nicht verschieden, bei aktivierter Option sind sie es.

9.2.7.4.2.3 Wortgrenzen

Diese Option wirkt sich auf die Erkennung literaler Token aus, nicht jedoch auf reguläre Ausdrücke. Bei aktivierter Wortgrenzen-Option gilt ein Token nur dann als erkannt, wenn es mit einer Wortgrenze beginnt und endet. An einer Wortgrenze in TETRA ist zumeist ein Zeichen beteiligt, das nicht alphanumerische ist und kein Unterstrich ist, also nicht zur Zeichenklasse \w gehört. Genau ist die Grenze durch drei Fälle definiert

das an das Token angrenzende Zeichen gehört nicht zu \w oder
das außenstehende Zeichen des Tokens gehört nicht zu \w oder
das Token bildet den Beginn bzw. das Ende des Eingabetextes

Beispiel:

in dem Text:

"sindbad der seefahrer"

hätten folgende Ausdrücke jeweils zei Wortgrenzen: "sindbad", "der" und "seefahrer".
Jeweils nur ein Wortgrenze hätten z.B.: "sind", "bad", see" und "fahrer".

An dem Beispiel erkennt man, dass es bei deaktivierten Wortgrenzen möglich ist, auch die interne Struktur einzelner literaler Worte zu analysieren.

Im Normalfall wird empfohlen die Option zu aktivieren, da es sonst zu falschen Erkennungen kommen kann. Ist bei deaktivierter Wortgrenze beispielsweise ein Token "end" definiert, so erkennt es fälschlich bereits den Anfang des Variablennamens in der folgenden Zeile

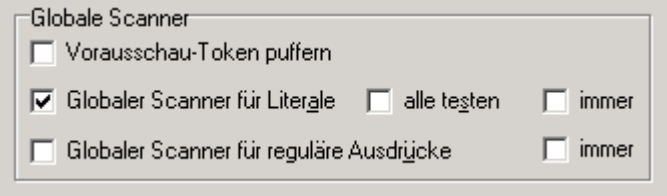
```
  endVar := 10;
end
```

9.2.7.4.2.4 Parameter und {{...}}

Hier wird eingestellt, ob nicht geklammerter Text des Parameterfeldes oder des Feldes für die den Token zugeordneten semantischen Aktionen interpretierbar oder exportierbar sein soll oder beides; d.h., ob sie intern ausgeführt oder in den generierten Code kopiert werden. Zugleich bestimmt diese Einstellung, wie semantische Aktionen, die in Paaren von gedoppelten geschweiften Klammern eingeschlossen sind behandelt werden.

9.2.7.4.2.5 Globaler Scanner

Auf der Seite "Parser/Scanner" der Projektoptionen gibt es drei Checkboxen durch die eine Feinjustierung der Scan-Prozesses vorgenommen werden kann.



Es wird empfohlen, die Default-Einstellungen beizubehalten. wenn man nicht die folgenden Ausführungen studieren will und spezielle Gründe findet, sie zu ändern. Als Standard ist nur die Verwendung des globalen Scanners für Literale aktiviert.

Die Pufferung der Vorausschau-Token kann die Ausführung eines Projektes beschleunigen, wenn in ihm von der Möglichkeit zur Vorausschau starker Gebrauch gemacht wird. Meist ist dies nicht der Fall.

Die anderen Einstellungen in dieser Box dienen der Kontrolle der Tokenmengen, die jeweils an der aktuellen Stelle der Grammatik getestet werden. Dadurch werden die Geschwindigkeit des Parsers und die Fehlerwahrscheinlichkeit beeinflusst. Größere Tokenmengen verlangsamen den Parser und erhöhen die Wahrscheinlichkeit, dass ein Token ermittelt wird, dass nicht zur Grammatik passt. Letzteres kann in bestimmten Fällen durchaus gewünscht sein.
Werden **keine globalen Scanner** verwendet, so wird beim Parsen - mit lokalen Scannern - immer nur nach den Token gesucht die gemäß der Grammatik aktuell folgen können Hingegen, wenn globale Scanner verwendet werden, so kann **immer** auf **alle Literale getestet** werden und auch **immer** auf alle **regulären Ausdrücke** getestet werden oder auch nur dann alle getestet werden, wenn überhaupt ein Literal/regulärer Ausdruck erwartet wird.

Anmerkung: Intern gibt es noch einen dritten Scanner, der lokal oder global sein kann: der **Scanner für die auszulassenden Zeichen**. Ob dieser Scanner lokal oder global ist, wird dadurch bestimmt, ob die auszulassenden Zeichen, in den lokalen Optionen gegenüber den globalen verändert sind oder nicht.

Etwas ausführlicher das Gleiche nochmal:

Den drei Scannertypen entsprechen drei Schritte, die bei der Ermittlung des jeweils nächsten Tokens durchgeführt werden. Ausgehend von der aktuellen Position im Quelltext wird geprüft,

1. ob auszulassende Zeichen folgen, dann
2. ob hierauf ein literales Token folgt oder,
3. ob ein mittels eines regulären Ausdrucks definiertes Token folgt

Diese drei Tests können entweder durch einen einzelnen globalen Scanner durchgeführt werden oder durch lokale Scanner. Die Verwendung eines globalen Scanners ist die traditionelle Methode, die bei allen bisherigen Parsergeneratoren angewendet wurde. Die Verwendung von lokalen Scannern beruht auf der Idee, dass zur Ermittlung des nächsten Tokens, nur diejenigen Kandidaten geprüft werden müssen, die tatsächlich zur aktuellen Alternative gehören. Zum Testen der folgenden Struktur beispielsweise ist es zunächst nur erforderlich zu entscheiden, ob ein a- oder ein b-Token an der aktuellen Textposition vorkommt.

```
( a | b ) c d
```

Ein lokaler Scanner testet daher genau dies. Ein traditioneller globaler Scanner hingegen prüft auf alle Token der Grammatik, also mindestens das a-, b-, c- und d-Token. Das Ergebnis ist in diesem Beispiel für beide Scannertypen identisch. Der Unterschied liegt in der Geschwindigkeit und dem Aufwand. Bei Verwendung lokaler Scanner ist die Geschwindigkeit des Scanprozesses gegenüber der der traditionellen Methode größer. Erkauft wird dieser Vorteil durch einen größeren Speicherbedarf der Menge an lokalen Scanner gegenüber dem Bedarf eines einzelnen globalen Scanners.

Auch das Ergebnis der Textanalyse kann durch die Wahl globaler bzw. lokaler Scanner beeinflusst werden. Bei Verwendung eines globalen Scanners ist die Wahrscheinlichkeit, das es Konflikte zwischen einzelnen Token gibt größer als bei Verwendung lokaler Scanner, in denen jeweils nur eine kleinere Anzahl von Token zueinander in Konkurrenz stehen. Dies ist der Grund dafür, warum es die zusätzliche Möglichkeit gibt, das **Testen auf die aktuell erwarteten Token einzuschränken**, auch wenn ein globaler Scanner für Literale verwendet wird.

Beispiel:

Text:	"int int"
Produktion:	"int" ID
Token ID:	\w+

Wenn stets alle literalen Token getestet werden, wird das zweite Vorkommen von "int" nicht als ID erkannt. Es wird vielmehr erneut das literale Token "int" erkannt. Der Text kann also nicht geparst werden. Dies ist erwünscht, wenn der Text einen C++-Code darstellen soll. Eine Variable darf nicht den Namen eines Variablen-Typs haben.

Text:	"Herr Herr"
Produktion:	"Herr" NAME
Token NAME:	\w+

Schauen sie ins Telefonbuch, und sie werden den Namen "Herr" darin finden. Die Anrede "Herr Herr" ist also durchaus korrekt. Sie wird nur erkannt, wenn nicht auf alle literalen Token getestet wird.

Faustregel:
Alle Literale sollten getestet werden, falls eine formalisierte Sprachen mit definierten Schlüsselworten, die an ausgezeichneten Positionen stehen, geparst werden soll. Andernefalls sollten nur die erwarteten Literale getestet werden. Nötigenfalls können auch die lokalen Optionen jeweils angepasst werden.

(Ein analoges Beispiel ist die SemText-Regel im CC2TT-Projekt.)

Konflikte, die durch die Verwendung eines globalen Scanners verursacht sind, können zur Laufzeit folgende Fehlermeldung hervorrufen:

Passendes aber nicht akzeptiertes Token: ...

Bei der Beschreibung dieser Meldung ist auch ein Beispiel angeführt.

Bei der Erläuterung der Vorausschau Produktionen wird ein weiteres Beispiel für die Auswirkungen der Scanner Optionen gegeben.

Durch Pufferung der Vorausschau-Token kann das Parsen beschleunigt werden, wenn Vorausschau-Produktionen verwendet werden. Wird in einer Produktion zunächst eine Vorausschau getestet, so werden die dabei ermittelten Token auf einen Stack gelegt und von dort wieder ausgelesen, wenn die ursprüngliche Produktion weiter ausgeführt wird.

Beispiel:

```
Prod1  ::= IF( Prod2() ) "a" "b" ...
Prod2  ::= "a" "b"
```

Bei erfolgreichem Test von *Prod2* sind die Token "a" und "b" bereits aus dem Text heraus gelesen.

Damit dieses Verfahren funktionieren kann, müssen einige Einschränkungen der sonst vorhandenen Möglichkeiten in Kauf genommen werden. Folgendes geht dann z.B. nicht mehr:

```
Prod1  ::= IF( Prod2() ) ID+ ...
Prod2  ::= "a" "b"
```

Oft ist zu empfehlen, anstelle von SKIP ist ANY zu verwenden. Eine mit *ANY+* in einer Vorausschau erkannte Tokenfolge, wird beim Auslesen des Stack korrekt als Folge der spezifischen Token reproduziert, die jeweils unter *ANY* subsumiert wurden.

In den semantischen Aktionen kann nicht auf Unterausdrücke der vom Stack geholten Token zugegriffen werden. Einschlüsse können nicht in Kombination mit dem Vorausschau-Puffer verwendet werden.

9.2.7.4.3 Start-Parameter

Hier können in zwei Felder die Parameter eingegeben werden, die bei Ausführung eines Projekts in der Arbeitsoberfläche der IDE von den Funktionen ConfigParam und ExtraParam geliefert werden. Bei der Ausführung des Projekts über den Transformations-Manager, im Kommandozeilenwerkzeug und für den generierten Code werden die hier gesetzten Parameter ignoriert und stattdessen die Parameter der entsprechenden Umgebung verwendet.

9.2.7.4.4 Einschlüsse (Kommentare)

Hier kann aus einer Auswahlbox eine Produktion zum Parsen von Einschlüssen - meist Kommentaren - gewählt werden. Wenn diese Produktion in den globalen Projektoptionen gesetzt ist, dann ist sie für alle Produktionen des Projekts gültig, d.h.auch für die gewählte Produktion selbst; so werden z.B. verschachtelte Kommentare korrekt geparst.

Beispiel:

```
CppComment ::= "/*" ( NUMBER | ID | "." | "-" )* "*/"
```

Wenn *CppComment* in den Projektoptionen zum Parsen von Einschlüssen gesetzt ist, wird auch folgender Kommentar geparst:

```
/* 1. Kommentar-Ebene /* 2. Kommentar-Ebene */ 1. Kommentar-Ebene */
```

Leider funktioniert folgende Produktion nicht, wenn Kommentare verschachtelt sind:

```
CppComment ::= "/*" ( SKIP | STRING )* "*/"
// ! diese Definition ist für verschachtelte Kommentare nicht geeignet
```

Der durch SKIP erkannte Text wäre:

```
1. Kommentar-Ebene /* 2. Kommentar-Ebene
```

Der Beginn des inneren Kommentars würde also übersprungen und das Ende des inneren Kommentars als Ende des gesamten Einschlusses interpretiert.

Ist die Verschachtelung nicht gewünscht, so kann in den lokalen Optionen von *CppComment* das Leerfeld für die Einschlüsse gesetzt werden. Ebenso kann für jede andere Produktion individuell eingestellt werden, welche Einschlüsse in ihnen getestet werden.

9.2.7.4.5 Kodierung

Bei den Einstellungen dieser Register-Seite sind einige Einschränkungen zu beachten. **Die Einstellungen können nicht vollständig im Debug-Modus reproduziert werden** und die Funktionen für Schreiben einer UTF-8 kodierten Ausgabe, sind nur im Transformations-Manager des TextTransformers und in dem zugehörigen Kommandozeilen-Tool implementiert. Sie sind nicht Teil des Codes, der mit der Professional-Version ausgeliefert wird.

Für den Quelltext und den Zieltext kann hier unabhängig eingestellt werden, wie sie codiert werden:

ANSI-Text
UTF-8

Wenn UTF-8 gesetzt ist, kann der Befehl *RedirectOutput* kann nicht benutzt werden. Durch diese Option wird RedirectOutput bereits einmal ausgeführt.

Weiter kann für Quell- und den Zieldateien unabhängig eingestellt werden, in welchem Modus sie geöffnet werden:

Text-Modus
Binär-Modus

Werden Dateien im **Binär-Modus** gelesen und geschrieben, so unterscheiden sich ihre Daten im Arbeitsspeicher nicht von denen auf der Festplatte.
Im **Text-Modus** hingegen werden unter Windows zur Behandlung der Zeilenumbrüche Konvertierungen durchgeführt. So wird z.B. das einfache Linfeed-Zeichen '\n' beim Schreiben mit einem Carriage-Return-Zeichen zu "\r\n" kombiniert.

Beispiel:

Nach Ausführung der Anweisung:

```
out << '\n';      bzw.: out << endl;
```

steht im Ausgabetext '\r\n'.

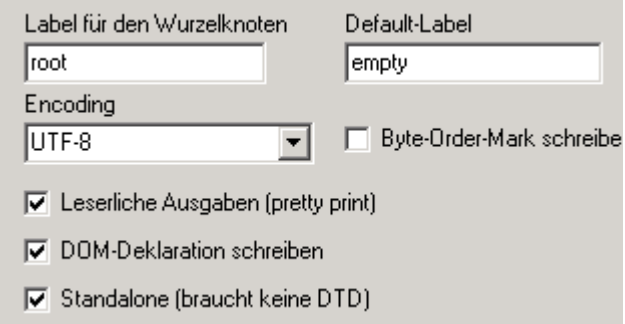
Im **Binär-Modus** erfolgt eine solche Umwandlung nicht. Dann könnte ein Windows-Zeilenumbruch explizit geschrieben werden als:

```
out << "\r\n"  bzw.: out << '\r' << endl;
```

Der Editor - i.U. zum Betrachter - ist nicht in der Lage Zeilenumbrüche aus einfachen Linefeed-Zeichen darzustellen. (Beide Arten von Zeilenumbrüchen könne aber gemeinsam durch das EOL Token erkannt werden.)

9.2.7.4.6 xerces DOM

In der oberen Hälfte der Dialogseite befinden sich die Optionen, die für ein alleinstehendes ("standalone") XML-Dokument gesetzt sein müssen. Ein derartiges Dokument hängt nicht von einer DTD ab.



Die angezeigten Optionen produzieren ein Dokument, das aussieht wie folgt:

```
<?xml  version="1.0"  encoding="UTF-8"  standalone="yes"  ?>
<root>
...
</root>
```

Label für den Wurzelknoten

Hier wird das tag definiert, dass die Wurzel des XML-Dokuments ausmacht. Im Beispiel wird es "root" genannt.

Default-Label

Jedes tag in einem XML-Dokument muss einen Namen haben. Dieser Name ist das Label einer *dnode*. Um die Analogie der Konstruktionsmöglichkeiten von Bäumen aus *node*'s und *dnode*'s zu bewahren, ist es erlaubt *dnode*'s ohne explizit definiertes Label zu verwenden. Solche *dnode*'s erhalten dann intern automatisch das Default-Label zugewiesen.

```
dnode dn("". "text");
```

erscheint im XML Dokument mit obigem Default-Label als:

```
<empty>text</empty>
```

Vor dem Aufruf des Parsers im generierten C++-Code, muss das Default-Label manuell gesetzt werden. Die CTT_DomNode Klasse hat hierfür eine statische Methode;

```
dnode::SetDefaultLabel(L"default_label");
```

Kodierung:

Xerces unterstützt zur Ausgabe der XML-Dokumente viele Kodierungen, die gleich diskutiert werden. In der Arbeitsoberfläche des TextTransformer wird aber nur der ANSI-Zeichensatz (Windows-1252) korrekt dargestellt. In der Entwicklungsphase eines Projekts sollte daher möglichst mit ANSI oder einer anderen 8-Bit Kodierung gearbeitet werden. Andernfalls erscheint die Meldung im Ausgabefenster:

Kodierung kann nicht in das Ausgabefenster der IDE geschrieben werden

Wenn eine Projekt über den Transformations-Manager, das Kommandozeilen-Werkzeug oder die Delphi-Komponenten ausgeführt wird, besteht keine Einschränkung für die Kodierung.

Einige der folgenden Bemerkungen zu den Kodierungen stammen von

http://xerces.apache.org/xerces-c/faq-parse.html

ASCII

ISO-8859-1 (aka Latin1)

Auf UNIX Systemen in Ländern, in denen westeuropäische Sprachen gesprochen werden, wird normalerweise ISO-8859-1 verwendet.

Windows-1252

Der Default-Zeichensatz auf Windows Systemen ist Windows-1252 (ANSI), und nicht ISO-8859-1. Während Xerces-C++ diese Windows Kodierung verarbeitet, ist die Wahrscheinlichkeit für die Portabilität der XML-Daten gering, da es von anderen XML-Verarbeitungswerkzeugen wenig unterstützt wird.

UTF-8

UTF-8 - wie UTF-16 - deckt den vollen Unicode-Zeichensatz ab, der alle wichtigen nationalen und internationalen Zeichen sowie Industrie-Zeichensätze umfasst. Diese Kodierung - ebenso wie UTF-16 - ist durch XML-Prozessoren besser unterstützt als alle übrigen Kodierungen.
Effizientes. utf-8 hat geringe Speicheranforderungen für Dokumente, die hauptsächlich aus Zeichen des lateinischen Alphabets bestehen.

UTF-16 (Big/Small Endian)

UTF-16 - wie UTF-8 - deckt den vollen Unicode-Zeichensatz ab, der alle wichtigen nationalen und internationalen Zeichen sowie Industrie-Zeichensätze umfasst. Diese Kodierung - ebenso wie UTF-8 - ist durch XML-Prozessoren besser unterstützt als alle übrigen Kodierungen.

UCS4 (Big/Small Endian)

EBCDIC code pages IBM037, IBM1047 and IBM1140 encodings
(Extended Binary Coded Decimals Interchange Code)

IBM1140
IBM037
IBM1047

Wenn EBCDIC kodierte XML-Daten erzeugt werden, ist IBM1140 die bevorzugte Kodierung. Die IBM037-Kodierung, auch ebcdic-cp-us genannt, ist nahezu identisch mit IBM1140, aber ihm fehlt das Euro Symbol.

Byte-Order-Mark (BOM) schreiben

Bei einigen Kodierungen wird eine Byte Order Mark (BOM, dt. „Bytereihenfolge-Markierung“) an den Anfang einer Datei gesetzt. Die Markierung gibt an, in welcher Reihenfolge die Bytes ausgewertet

werden müssen, falls einzelne Zeichen aus mehreren Bytes bestehen, wie in UTF-16.

Das BOM wird nur für folgende Kodierungen geschrieben

- UTF-16
- UTF-16LE
- UTF-16BE
- UCS-4
- UCS-4LE
- UCS-4BE

Bei UTF-8 kann eine solche Markierung optional dazu verwendet werden, die Datei als UTF-8 kodiert zu kennzeichnen. Es ist aber nicht möglich für UTF-8 die automatische Voranstellung einer BOM's einzustellen, da xerces dies nicht unterstützt. Die BOM kann aber durch folgenden Code im Programm dem Dokument vorausgeschickt werden:

```
out << char(0xEF) << char(0xBB) << char(0xBF);
WriteDocument();
```

Wird eine UTF-8 kodierte Datei als ANSI-Datei gelesen, erscheint diese Marke als Zeichenfolge: ï»¿.

Kodierung	Bytefolge
UTF-8	EF BB BF
UTF-16 Big Endian	FE FF
UTF-16 Little Endian	FF FE

Leserliche Ausgabe (pretty-print)

Durch Einfügung von Zeilenumbrüche und Leerzeichen wird das Dokument in einer vom Menschen gut lesbaren Form - *pretty-print* -erzeugt.

Wenn diese Option gesetzt ist und das Dokument mit *WriteDocument* ohne Parameter geschrieben werden soll, ist es für einige Kodierungen - z.B. für UTF-16 - erforderlich, die Option für eine binäre Ausgabe zu setzen.

DOM-Deklaration schreiben

In obigem Beispiel ist die Zeile:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
```

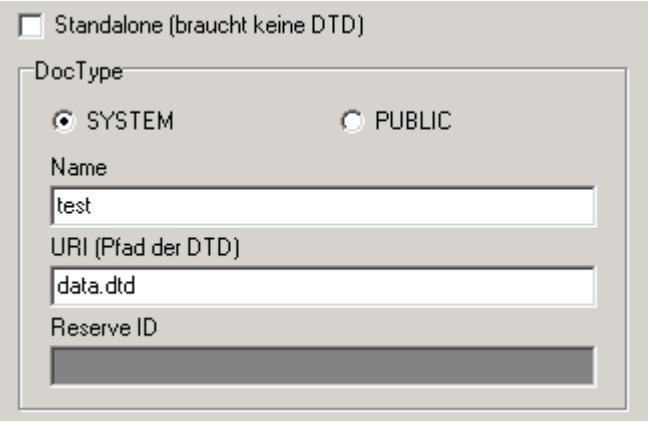
die Deklaration des Dokuments. Soll sie ganz weggelassen werden oder im Programm explizit formuliert werden - nur für UTF-8 - , so kann die Checkbox deaktiviert werden.

Standalone

Wenn das erzeugte Dokument nicht von einer DTD abhängen soll, so sollte diesen Box aktiviert sein und umgekehrt.

9.2.7.4.6.1 DTD

In der unteren Hälfte der Dialogseite befinden sich die Optionen, die für ein XML-Dokument gesetzt sein müssen, das von einer DTD (document type definition) abhängt.



Die angezeigten Optionen produzieren ein Dokument, das aussieht wie folgt:

```
<?xml  version="1.0"  encoding="UTF-8"  standalone="yes"  ?>
<!DOCTYPE  test  SYSTEM  "data.dtd">
<root>
...
</root>
```

Mit der DTD "data.dtd" wird geprüft, ob das Dokument gültig ist. Diese Prüfung erfolgt aber erst beim Neuladen des Dokuments in einem validierenden XML-Prozessor und nicht schon im TextTransformer.

Falls eine öffentliche DTD verwendet werden soll, muss der entsprechende Radio-Schalter angeklickt werden und die Deaktivierung des unteren Feldes für die Reserve-ID wird aufgehoben. Die Angabe einer Reserve-ID ist dann erforderlich.

9.2.7.4.7 Warnungen/Fehler

Die Erzeugung von Warnungen kann unterdrückt oder aktiviert werden.
Dies sind die Warnungen

Knoten ist löschbar
Knoten ist Anfänger und Nachfolger löschbarer Strukturen
xState-Parameter für Klassenfunktionen
Überlappende Systeme

Maximaler Stack

9.2.7.4.7.1 Maximale Stack-Größen

Die **erste Größe** bezieht sich auf den Stack der Produktionen des Hauptparsers.
Durch diesen Wert wird die maximale Größe des internen Stacks begrenzt. Die Begrenzung verhindert unendliche Schleifen, wie sie bei Linksrekursionen auftreten können. Der Standard-Wert 100 reicht meist aus.

Anm.: Der interne Stack ist dabei größer als der angezeigte da er sämtliche Verzweigungen enthält und nicht nur die Verzweigungen in Unterregeln.

Die **zweite Größe** bezieht sich auf den Stack der Produktionen bei einer Vorausschau. Die Begrenzung dieses Stacks ist wichtig, um bei eventuellen zirkulären Aufrufen einen Absturz des Systems zu vermeiden.

9.2.7.4.8 Code-Erzeugung

Diese Optionen sind nur für Benutzer der Professional-Version von Interesse

const
Wide-Character-Regex
Sämtlichen Code nur kopieren
Zeichen und Schrittweite der Einrückungen
Plugin-Typ
Template Parameter für Plugin-Zeichentyp

9.2.7.4.8.1 const

const

Die Aktivierung dieser Option macht nur in der Professional Version Sinn, wenn C++-Code exportiert werden soll. In den interpretierte Anwendungen der Standard-Version bedeutet sie ein unnötige Einschränkung der Programmiermöglichkeiten.

Wenn die const-Option durch ein Häkchen in der Checkbox aktiviert ist, werden sämtliche Funktionen (Produktionen, Token-Aktionen und Interpreter-Funktionen) des generierten Parsers zu konstanten Funktionen.

Beispiel.:

```
void CCalcParser::Expression(double xd) const;
```

Damit wird sicher gestellt, dass diese Funktionen die Daten der Parserklasse nicht verändern können. Dies ist z.B. wichtig, wenn Multithreaded Anwendungen erzeugt werden sollen.

Bei aktivierter const-Option sind alle Operationen verboten, die den Zustand der Parserklasse

verändern würden. Dazu gehören z.B. auch die Operationen, die den Cursor einer mstrstr Klassenvariable verschieben würden (findKey, gotoNext etc.). Es sit jedoch möglich mstrstr -Referenzen als Klassenvariablen zu benutzen. Auch Quell- und Zielverzeichnisse und der Einrückungsstack können nur als entsprechende Referenzvariablen verwendet werden. Die Klassenschablone muss entsprechend modifiziert werden.

9.2.7.4.8.2 "Wide"-Zeichen verwenden

Wenn die Option zur Verwendung von "Wide" (= "Breite") Zeichen aktiviert ist, werden Parserklassen erzeugt, die Texte verarbeiten können, die Zeichensätze verwenden, die mehr als die 256 ASCII-Zeichen enthalten (Unicode). Ein einzelnes Zeichen wird dann nicht durch ein Byte sondern durch zwei Bytes im Speicher repräsentiert; statt mit *char*, *std::string* und *boost::regex* mit den entsprechenden Datentypen: *wchar_t*, *std::wstring* und *boost::wregex* gearbeitet.

Die Parserklasse im generierten Code ist bei Verwendung dieser Option von

```
CTT_Parser<wchar_t>
```

abgeleitet. Sämtliche Texte und Stringfunktionen das Parsersystems basieren dann auf dem *wchar_t* -Typ.

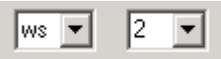
Das "diagnostische" System wird aber durch den *wchar_t* Template-Parameter nicht verändert: Fehlertexte werden weiterhin als *char*-Strings ausgegeben und Meta-Funktionen, wie *ProductionName* etc. geben ebenfalls nur std::string zurück. Bei einer unglücklichen Vermischung der Systeme kann es vorkommen, dass der generierte C++-Code wegen inkompatibler String-Operationen nicht kompiliert.

9.2.7.4.8.3 Sämtlichen Code nur kopieren

Das Setzen dieser Option bewirkt, dass der Code, der den semantischen Aktionen in eine generierte Parserklasse übernommen wird, schlicht kopiert wird.
Im Normalfall wird die Option nicht gesetzt. Dann wird der interpretierbare Anteil aus der geparsten Form rekonstruiert, wobei es zu einer neuen Formatierung und der Umformung einiger Konstrukte kommt. Die exportierbaren Teile werden in diesem Fall ebenfalls verändert.
Eine Umformung ist nötig für:

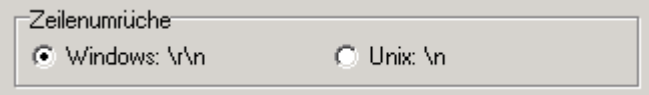
- Aufrufe von Klassenfunktionen, in die ein zusätzlicher *xState*-Parameter eingefügt wird
- Abgekürzte Schreibweisen, die nur im Interpreter gültig sind, werden ersetzt: out -> xState.out(); indent -> xState.indent(), format -> boost::format...
- Einfügungen (mit *add*) von Klassenfunktionen in Funktionstabellen. Hier wird der zweite String-Parameter, der den Funktionsnamen angibt, in einen Zeiger auf eine Klassenfunktion umgewandelt
- Falls WideChar-Parser generiert werden, werden einige Ersetzungen vorgenommen: "Hi" -> L"Hi"; format -> boost::wformat;

9.2.7.4.8.4 Zeichen und Schrittweite der Einrückungen



Einrückungen im generierten Parser-Code können wahlweise durch Leerzeichen **ws** oder durch Tabulatoren **tab** erfolgen. Das Maß der Einrückung wird durch die Anzahl dieser Zeichen bestimmt, die an den jeweiligen Positionen eingefügt werden.

9.2.7.4.8.5 Betriebssystem



Für das Windows Betriebssystem werden die generierten Codedateien mit "\r\n"-Zeilenumbrüchen versehen und für Unix-Systeme mit "\n"-Zeilenumbrüchen.

Anmerkung: Diese Einstellung hat keinen Einfluss auf das Verhalten des Parsers.

9.2.7.4.8.6 Plugin-Typ

Für einen Parser kann ein Plugin definiert werden, das außerhalb des Parsers für einen Parserdurchlauf spezifisch initialisiert werden kann, und das dann der Startregel übergeben und innerhalb des Parsers verwendet werden kann, z.B. auch um Resultate zu speichern. Durch die Verwendung eines Plugins sind dynamische Daten auch in const-Parsern für multithreaded Anwendungen möglich.
Der Zeiger auf den Plugin-Typ wird in der Parserzustandsklasse gesetzt. Standardmäßig ist dies:

CTT_ParseStatePlugin

Werden in dem Projekt dnode Knoten verwendet, muss

CTT_ParseStateDomPlugin

gewählt werden. In diesem Fall muss die Xerces Bibliothek zum erzeugten Code gelinkt werden.

Erfordert der Plugin-Typ einen Template Parameter für den Zeichentyp, so kann dies durch die nachfolgende Option angegeben werden.

Wird der Interface-Methode kein solcher Zeiger übergeben, wird eine lokale Instanz des Plugins erzeugt. CTT_ParseStatePlugin enthält alle Daten, die für die Plugin-Methoden erforderlich sind.

Benutzerdefinierte Plugins müssen von CTT_ParseStatePlugin oder von CTT_ParseStatePluginAbs abgeleitet sein.

Der Zeiger auf den Plugin-Typ ist Template-Parameter der Parserzustansklasse und als typedef auch in der Parserklasse selbst verfügbar. So ist der vollständige Typ des Plugins innerhalb des Parsers bekannt und es ist kein Typecast nötig, um auf die Daten und Funktionen des Plugins zugreifen zu können.

9.2.7.4.8.7 Template Parameter für Plugin-Zeichentyp

Ist diese Option gesetzt, so wird an die Bezeichnung des Plugin-Typs der Template-Parameter "< char_type >" angefügt. Ist beispielsweise der Default-Plugin-Typ gesetzt:

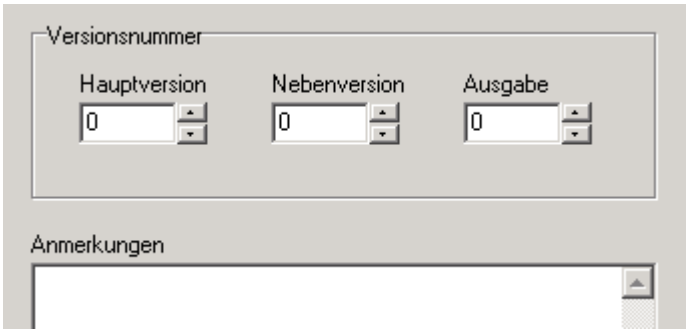
CTT_ParseStatePlugin

so wird daraus im erzeugten Quellcode

CTT_ParseStatePlugin< char_type >

9.2.7.4.9 Versionsinformationen

Auf der Seite mit dem Reiter *Version* können Versionsinformationen zu einem Projekt gespeichert werden.



9.2.7.5 Lokale Optionen

Die Projekteinstellungen, die zunächst für alle Regeln gelten, können für einzelne Produktionen durch lokale Optionen überschrieben werden. Der Menüeintrag für die lokalen Optionen befindet sich im Hautmenu: Projekt, das nur auf der Produktionenseite sichtbar ist.

9.2.7.5.1 Lokale Optionen

Die Projekteinstellungen, die zunächst für alle Regeln gelten, können für einzelne Produktionen überschrieben werden. Der Menüeintrag für die lokalen Optionen befindet sich im Hauptmenü: Projekt, das nur auf der Produktionenseite sichtbar ist. Die lokalen Einstellungen einer Produktion wirken sich nicht auf die Arbeitsweise der in ihr aufgerufenen Produktionen aus.

Bevor die einzelnen Optionen eingestellt werden können, müssen die lokalen Optionen auf der ersten Tabellenseite aktiviert werden. Sind für eine Produktion lokale Optionen aktiv, die die globalen

Projektoptionen überschreiben (i.U. zu rein lokalen Optionen), so wird ihr Knoten (nach Kompilierung) mit einem roten Häkchen versehen.

Die Option zur Verwendung globaler Scanner ist hier eingeschränkt:

Globaler Scanner für Literale, falls möglich
Globaler Scanner für Regexe, falls möglich

Die Aktivierung globaler Scanner in den lokalen Optionen wird nur wirksam, wenn die *Groß-/Kleinschreibung*-Option in den lokalen und globalen Optionen gleich gesetzt sind.
Eine lokale Deaktivierung der globalen Scanner ist immer möglich.

Werden Optionen für den Scanner lokal verändert, so wirkt sich diese Änderung aus, sobald innerhalb der Produktion ein neues Token ermittelt wird. Dies geschieht sobald ein Token in der Produktion akzeptiert wird oder, wenn das letzte Token einer in der Produktion aufgerufenen Produktion akzeptiert wird. Im letzten Fall wird in der aufrufenden Produktion der Nachfolger der aufgerufenen Produktion ermittelt.

Beispiel:

```
Prod1 ::= Prod2 "c"
Prod2 ::= "a" "b"
```

Wenn in *Prod1* lokal das Leerzeichen als auszulassen gesetzt sind, nicht aber in *Prod2* und in beiden Produktionen die Worttrennung deaktiviert ist, so wird die erste Reihe von Texten korrekt geparst, der zweite wird nicht geparst. Ob der dritte Text geparst wird hängt von der vorhergehenden Produktion ab.

1: "ab c", "abc" abc ", ab c "
2: "a bc"
3. " abc"

Anmerkung: Die **ausgelassenden Zeichen** werden über alle Tests akkumuliert.Wenn in der aktuellen Produktion kein Nachfolger für ein Token gefunden wird, wird der Nachfolger der Produktion selbst gesucht. Diese Suche startet dann nach den bereits ausgelassenen Zeichen.

Neben den in den Projektoptionen beschrieben Punkten gibt es eine zusätzliche Option:

Interfacemethode erzeugen

Diese Option lässt sich direkt in einer Checkbox in der Werkzeugleiste der Produktionen-Seite einstellen.

Ist diese Option aktiviert, so wird für die Produktion ein gesonderter Scanner erzeugt, der die Menge der Token umfasst, mit denen die Regel beginnen kann. Sind mehrere Produktionen kompiliert, für die die Interfaces erzeugt wurden, so kann zwischen ihnen während der Transformation eines Textes interaktiv gewechselt werden. Die Interface-Scanner testen dann unmittelbar nach einem solchen Wechsel, ob der jeweils aktuelle Text zu der neu gesetzten Startregel passt.

Anmerkung:

Werden die Produktionen mit Alle Skripte parsen kompiliert, hat dies den Effekt, als sei die Option zum Erzeugen eines Interfaces für alle Produktionen gesetzt. Automatisch wird für jede Produktion ein Scanner erzeugt, der das erste Token eines gewählten Textabschnitts testen kann, so dass zwischen sämtlichen Produktionen ein interaktiver Wechsel möglich ist.

Im Normalfall wird ein Parser über ein Interface zu seiner Startregel aufgerufen. Wenn eine entsprechende Interfacemethode erzeugt wird, ist es aber auch möglich andere Regeln direkt aufzurufen.
So werden z.B. im TextTransformer die Texte für die Parameter einer Produktion mit einer Unterregel des Produktionen-Parsers in die interpretierbaren und exportierbaren Teile zerlegt. Um die Unterregel direkt aufrufen zu können, musste für sie ein Interface erzeugt werden.

9.2.8 Menü: Fenster

Verteilt auf eine Vielzahl von andockbaren Fenstern werden im TextTransformer eine Menge von Informationen dargestellt. Je nach gerade anstehendem Arbeitsschritt und Art des Projekts wird ein bestimmter Teil davon benötigt. Die jeweils optimale Anordnung der Fenster hängt von der technischen Ausstattung (z.B. der Bildschirmauflösung) und den individuellen Vorlieben des Benutzers ab Deshalb gibt es die Möglichkeit spezielle Fenster-Layouts zu erstellen und abzuspeichern, um sie bei Bedarf wieder zur Verfügung zu haben.

Für die beiden grundlegenden Arbeitsschritte, den Bearbeiten eines Projekt und der Fehlersuche (dem Debuggen) gibt es vorgefertigte Default-Layouts:

EditDefault.ds
DebugDefault.ds

Wenn der TextTransformer zwischen Edier- und Debug-Modus wechselt wird automatisch auch das Layout gewechselt. Auf der Layouts-Seite der Benutzer-Einstellungen ist es möglich, andere Layouts für die beiden Programm-Modi zu wählen.

Die Menü-Einträge im einzelnen sind:

Fensterliste
Layout anpassen
Layout speichern
Default Edier-Layout wiederherstellen
Default Debug-Layout wiederherstellen
Fenster

Ein (englisches) Viedeo zur Anpassung eines Layouts gibt es unter:

http://www.texttransformer.com/Videos_en.html

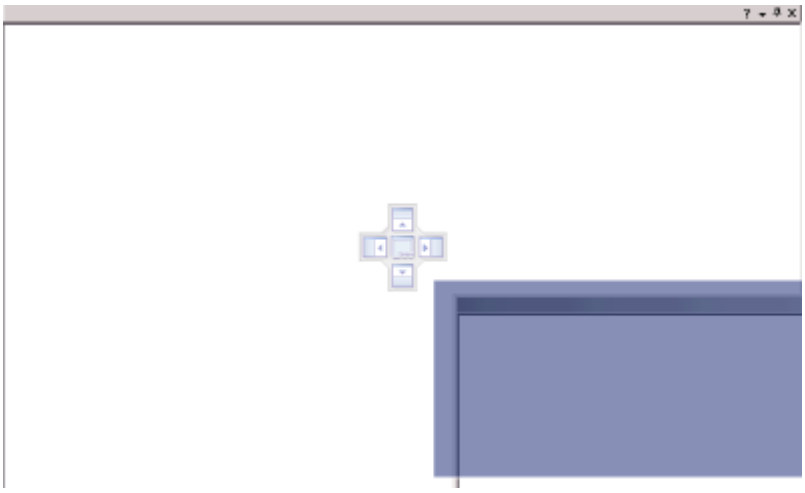
9.2.8.1 Andockbare Fenster

Das Layout für die Fenster kann im TextTransformer den eigenen Wünschen angepasst und als Schablone abgespeichert werden. Diese Möglichkeit ist besonders wertvoll um auch bei kleineren Bildschirmauflösungen die jeweils benötigten Fenster gleichzeitig anzuzeigen.

Die relative Größe der Fenster zueinander kann meist mit der Maus verändert werden, indem sie über die gemeinsame Trennlinie geführt wird, wobei der Mauszeiger die Form

annimmt. Bei gedrückter linker Maustaste kann nun die Trennlinie auf die gewünschte Position gezogen werden, wo sie dann nach Loslassen der Maustaste verbleibt.

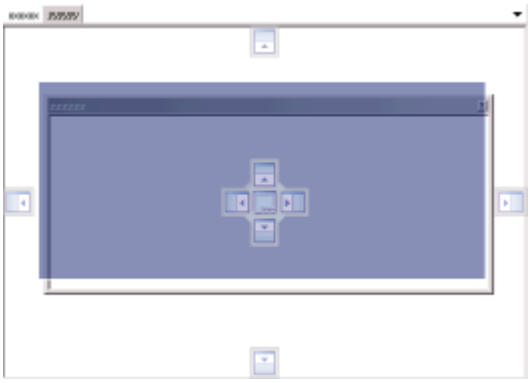
Ein Fenster kann einfach abgedockt werden, indem man auf die obere Leiste doppel-klickt oder indem man ihn bei gedrückter linker Maustaste wegzieht. Dadurch wird das Fenster zu einem normalen frei schwebenden Fenster. Ein solches Fenster kann schnell wieder an die vorherige Position gebracht werden, indem man erneut einen Doppelklick auf den oberen Rand ausführt. Wenn die Position des Fensters geändert werden soll, so kann man es einfach an die gewünschte Stelle ziehen. Wenn man sich dabei über einem anderen Fenster befindet, zeigt der TextTransformer eine Andock-Auswahl. Mit ihrer Hilfe lässt sich bestimmen, wo das Fenster angedockt werden wird, wenn man es los lässt:.



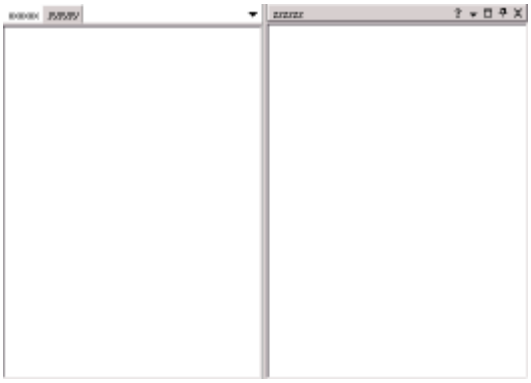
Um das Fenster an den linken, oberen, rechten oder unteren Rand des unten liegenden Fensters anzudocken, wird der Maus-Zeiger auf das  ,  ,  oder  Icon in der Auswahl bewegt und dann die Maustaste losgelassen. Auch, wenn das Fenster angedockt ist, hat es noch immer eine Leiste mit einem Namen links und den Schaltern rechts. Durch Ziehen an der oberen Leiste kann es erneut abgedockt werden.

Wenn das  Icon in der Andock-Auswahl gewählt wird, werden das obere und das untere Fenster an der gleichen Stelle angedockt sein. In diesem Fall werden die Fenster zu einer Register-Seite kombiniert. Es können auch noch weitere Fenster in das Register eingefügt werden. Um ein Fenster aus diesen Register zu entfernen, genügt wieder ein entsprechender Doppelklick oder das Ziehen des Fensters.

Wenn ein freies Fenster über eine Register-Seite gezogen wird, so werden neben der schon bekannten Andock-Auswahl im Zentrum der Register-Seite auch noch Wahlmöglichkeiten an den Rändern angeboten.

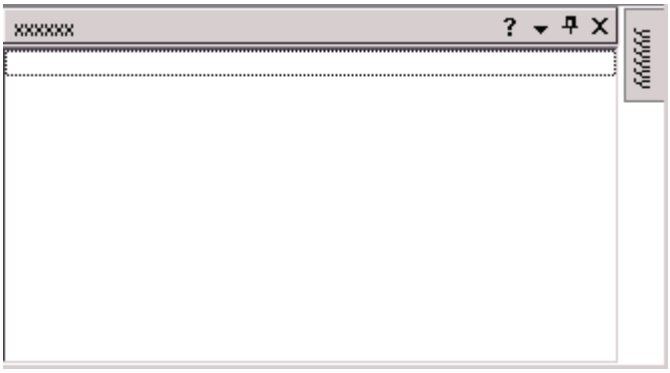


Wird ein von ihnen ausgewählt, so wird das zuvor freie Fenster relativ zur gesamten Register-Seite angedockt.



Wird eine der Möglichkeiten der zentralen Auswahl gewählt, so wird die freie Seite in dem Fenster angedockt, das aktuell in der Register-Seite im Vordergrund ist.

Ein angedocktes Fenster kann automatisch versteckt werden. D.h. es kann an einer Seite des Fensters, in dem es angedockt ist minimiert werden, so dass nur der Titel sichtbar bleibt, so wie das Fenster "yyyyyy" in der Abbildung.



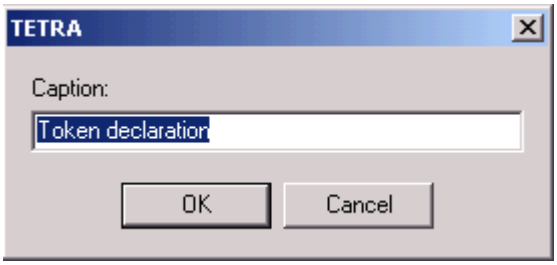
Um ein Fenster zu verstecken wird der Schalter in der Leiste angeklickt.
Um ein automatisch verstecktes Fenster anzuzeigen, wird der Mauszeiger auf den Titel bewegt oder der Titel direkt angeklickt.
Um ein verstecktes Fenster wieder sichtbar zu machen, wird auf den Schalter in der Titelleiste geklickt.

Nach Klick mit der rechten Maustaste wird folgendes Kontext-Menü angezeigt:

Umbenennen Ruft die Titel-Dialogbox auf, wo die Seite neu benannt werden kann.
Hilfe Den Punkt Hilfe gibt es nur für Einzelseiten. Über ihn wird ein Beschreibung der Seite erhalten. Sind mehrere Fenster zu einer Register-Seite kombiniert, gibt es keinen Hilfe-Eintrag.

9.2.8.1.1 Titel-Dialogbox

Die Titel-Dialogbox wird verwendet, um den Titel eines Fensters oder Registers zu ändern. Um diesen Dialog aufzurufen klickt man mit der rechten Maustaste auf den Titel bzw. das Register des entsprechenden Fensters und wählt dann **Rename** im **Kontext-Menü.**.



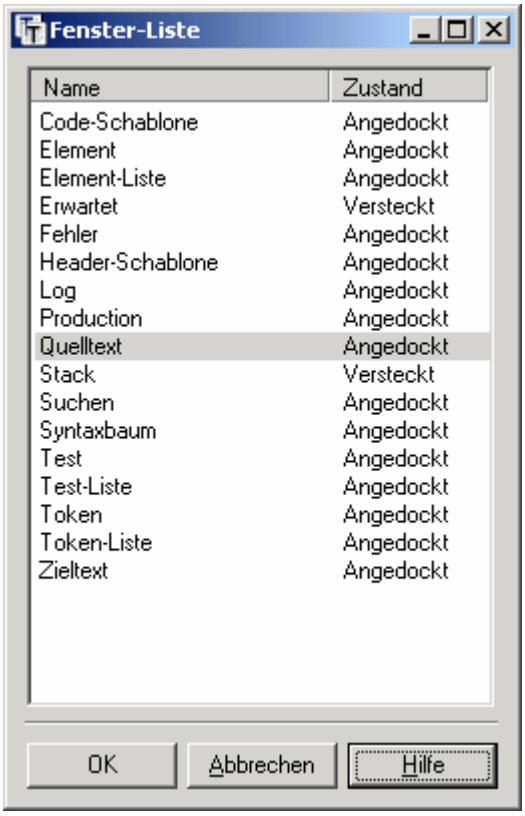
Siehe auch

Andockbare Fenster

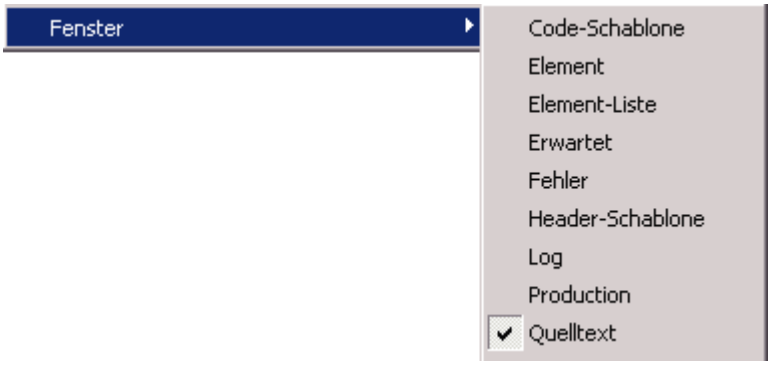
9.2.8.2 Fensterliste

Über die Fensterliste kann schnell zu Fenstern navigiert werden, die aktuell verdeckt oder geschlossen sind. Die Fensterliste gibt es gleich in doppelter Form:

1. als Dialog, der auch mit dem Schalter  in der Werkzeugleiste aufgerufen werden kann.



2. als Unter-Eintrag im Fenster-Menü, über den das gewünschte Fenster sehr schnell geöffnet werden kann.



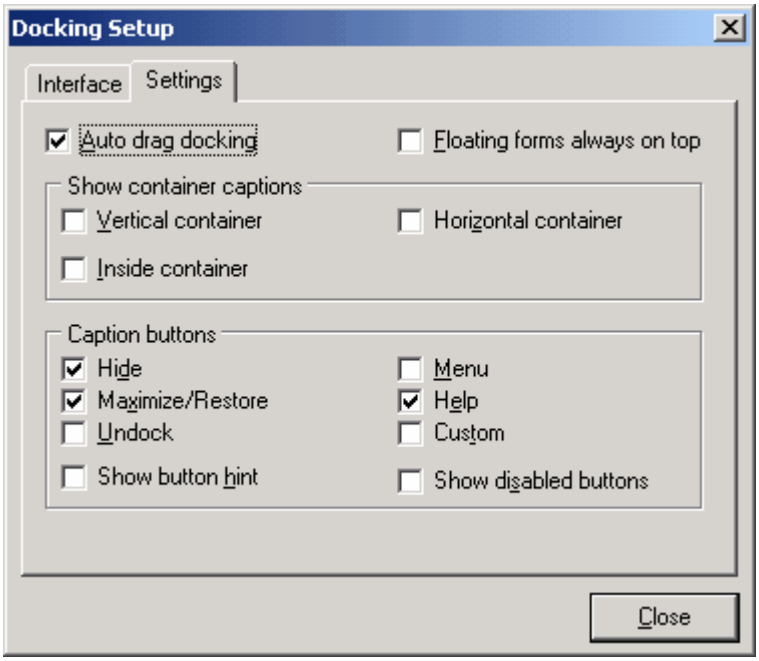
Wenn auf einen Eintrag in der Liste geklickt wird, wird das entsprechende Fenster angezeigt.

Folgende Fenster stehen zur Auswahl:

Code-Schablone
Element
Element-Liste
Erwartet
Fehler
Header-Schablone
Log
Produktion
Quelltext
Stack
Suchen
Syntaxbaum (= Produktionen-Liste)
Zieltext
Test
Test-Liste
Token
Token-Liste

9.2.8.3 Layout anpassen

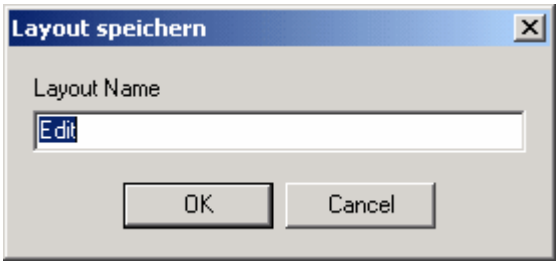
Der Dialog zum Anpassen des Layouts steht nur in der Professional Version zur Verfügung und nur mit englischer Beschriftung. Mit dem Dialog lassen sich z.B. zusätzliche Leisten einblenden, die das Verschieben von Fenster-Gruppen erleichtern. Die zusätzlich angezeigten Elemente gelten nur für die aktuelle Sitzung und werden nicht gespeichert.



9.2.8.4 Layout speichern

Das aktuelle Layout kann über den Eintrag *Layout speichern* im Fenster-Menü oder über den Schalter  in der Werkzeugleiste gespeichert werden.

Zunächst erscheint eine Dialogbox in die ein Name für das Layout eingegeben werden kann. Als Default ist der Name des aktuellen Layouts gesetzt.



Wenn *Ok* gedrückt wird, wird das Layout im Ordner DATEN-VERZEICHNIS\Settings mit angehängter Erweiterung ".ds" gespeichert.

Sämtliche Layouts in DATEN-VERZEICHNIS\Settings werden in der Auswahlbox links neben dem Schalter zum Speichern angezeigt:

Wenn ein Eintrag dieser Box ausgewählt wird, wird das entsprechende Layout geladen und die Fenster werden ihm entsprechend angeordnet.

.

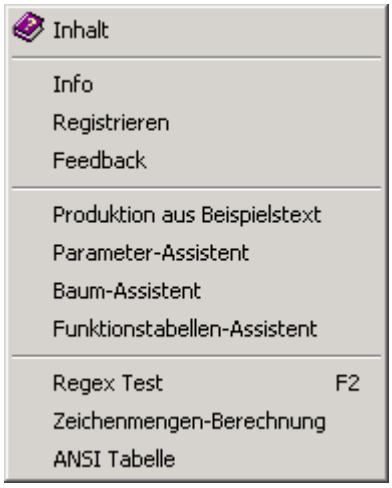
9.2.8.5 Default Layout wiederherstellen

Bei der Installation des TextTransformers werden die Layouts EditDefault.ds und DebugDefault.ds in DATEN-VERZEICHNIS\Settings geschriebenen. Diese Layouts sind für eine Bildschirmauflösung von 800 x 600 Pixel gemacht und sollten den eigenen technischen Gegebenheiten und Bedürfnissen angepasst werden. Um gefahrlos mit diesen Layouts experimentieren zu können, gibt es eine Sicherheitskopie in PROGRAMM-VERZEICHNIS\bin. mit den beiden Einträgen Hauptmenü Fenster:

Default Edier-Layout wiederherstellen
Default Debug-Layout wiederherstellen

können die Sicherheitskopien automatisch in den *Settings*-Ordner zurückkopiert werden.

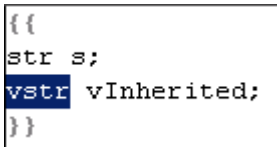
9.2.9 Menü: Hilfe



Über das Menü Hilfe ist nicht nur diese Hilfedatei und die Registrierung aufrufbar, sondern auch drei Hilfstools sowie Eine Reihe von Assistenten:

Regex Test
Zeichenmengen-Berechnung
ASCII Tabelle

Die Hilfe zu einzelnen Anweisungen kann direkt aufgerufen werden, indem das betreffende Schlüsselwort markiert wird und dann die Taste F1 gedrückt wird.



Die Hilfe zu Fehlermeldungen kann direkt aufgerufen werden, indem die entsprechende Zeile in der Fehlerliste einfach angeklickt wird und dann die Taste F1 gedrückt wird.

9.2.9.1 Feedback

Falls sie Probleme, Anmerkungen oder Ideen zum TextTransformer haben, ist ihr Feedback erwünscht. Bitte zögern sie nicht ein Mail zu schicken an

dme@texttransformer.de

Es ist empfehlenswert, diese Schablone für ihre Mail zu vewenden, da sie wichtige Informationen über die Version des TextTransformers und über ihr System enthält. Bitte Schreiben sie ihre Anmerkungen in die folgenden Zeilen und kopieren sie dann den gesamten Text in ihr E-Mail Programm.

Ihr Feedback wird ausschließlich zur Verbesserung des TextTransformers verwendet. Ihre Informationen werden nicht an Dritte weitergegeben und sie erhalten keine unerwünschten Werbeemails. Lediglich auf dieses Feedback erhalten sie eine adäquate Antwort oder einen Dank.

9.2.9.2 Assistenten

Eine Reihe von Assistenten machen die Arbeit mit TextTransformer einfacher. Bis auf den Assistenten für neue Projekte sind sie über das Hilfe-Menü aufzurufen.

Neues Projekt Assistent
Produktion aus einem Beispielstext erzeugen
Parameter-Assistent
Baum-Assistent
Funktions-Tabellen-Assistent

Die Assistenten bieten auf jeder ihrer Seite eine Anzahl von Optionen, die am oberen Rand der Seite erklärt werden. Je nach Auswahl werden mögliche weitere Seiten erreichbar, zu denen mit dem ***Next*** Schalter gegangen werden kann.

Die Operationen der Assistenten können nicht rückgängig gemacht werden! Bitte speichern sie ihr Projekt bevor sie Assistenten benutzen und laden sie das Projekt neu, falls ein Fehler auftrat.

Die Anwendung des Baum- und des Funktions-Tabellen-Assistenten wird für den Java-Parser ausführlich beschrieben.

9.2.9.2.1 Neues Projekt Assistent

Beim Anlegen eines neuen Projekts erscheint automatisch ein Assistent, der dabei hilft, grundlegende Einstellungen, je nach geplantem Projekt-Typ vorzunehmen. Dieser Assistent kann sogar vollständige Projekte für kleine Aufgaben erstellen, die in Ihrer täglichen Arbeit anfallen können.

Folgende Projekt-Typen stehen zur Auswahl:

Mehrfach-Ersetzung
- Mehrfach-Ersetzung von Worten
- Mehrfach-Ersetzung von Zeichen

Zeilen umschreiben
- CSV-Assistent
- Zeilenparser aus Beispielstext erzeugen

Kopf/Kapitel/Fuß-Rahmen

Neues Projekt ohne Vorgaben

Die Beispiels-Texte und -Projekte die auf den folgenden Seiten verwendet werden, sind zu finden im Verzeichnis:

..\TextTransformer\Beispiele\Assistenten

9.2.9.2.1.1 Mehrfach-Ersetzung von Worten

Mit dem TextTransformer können Projekte erstellt werden, die ähnlich arbeiten, wie die Funktion zum Suchen-und-Ersetzen in Textverarbeitungsprogrammen. Im Unterschied zur letzteren können mit dem TextTransformer ganze Stapel von Dateien bearbeitet werden und es können in jeder dieser Dateien viele Ersetzungen gleichzeitig vorgenommen werden. Der hier beschriebene Assistent unterstützt sie bei der Erstellung derartiger Projekte.

Dazu müssen lediglich in einer Tabelle die Liste von Such- und Ersetzungsausdrücke erstellt werden.
Die Eingabe der Werte in die Tabelle geschieht genauso wie bei den Tabellen anderer Assistenten.

Wenn sie alle Wortpaare eingegeben haben und den Bestätigen-Schalter betätigt haben, durch den

der Edier-Modus verlassen wird, bekommen sie auf der nächsten Seite den Text einer Produktion angezeigt. Wortersetzungs-Projekte bestehen aus nicht mehr als nur dieser einfachen Produktion.

```
(
    "Hello"     {{ out << xState.str(-1) << "Hallo";}}
  | "World"     {{ out << xState.str(-1) << "Welt";}}
  | WORD        {{ out << xState.copy(); }}
  | PUNCT       {{ out << xState.copy(); }}
)*

WORD ::= [^[:space:][:punct:]]+
PUNCT ::= [[:punct:]]+
```

Wenn sie nun den Assistenten beenden, ist das Projekt fertig für die Anwendung.

Die Ersetzungsliste kann abgespeichert und später neu geladen werden. Zumeist wird es aber sinnvoller sein Änderungen und Ergänzungen direkt im erzeugten Projekt vorzunehmen.

Im Verzeichnis:

..\TextTransformer\Beispiele\Assistenten

befinden zum Experimentieren sich zwei Listen mit englischen und deutschen Filmtiteln:

Film_en_ge.txt und Film_ge_en.txt

9.2.9.2.1.2 Mehrfachersetzung von Zeichenketten

Die Mehrfachersetzung von Zeichenketten ist der Mehrafachersetzung von Worten sehr ähnlich. In Projekten zur Mehrfachersetzung von Zeichenketten können aber auch Teile von Worten oder andere Textabschnitte ersetzt werden. Die Anzahl der Erstzungsmöglichkeiten ist hier aber begrenzt. (Eine Zahl unter Hundert sollte keine Probleme bereiten.)

Beispiel einer erzeugten Produktion:

```
(
    "Ha"     {{ out << xState.str(-1) << "He";}}
  | "elt"    {{ out << xState.str(-1) << "orld";}}
  | SKIP     {{ out << xState.copy(); }}
)*
```

Die Wortgrenzen-Option ist hier deaktiviert.

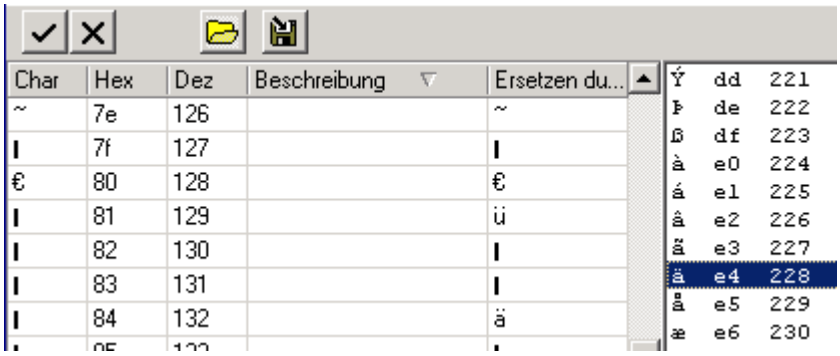
9.2.9.2.1.3 Mehrfach-Ersetzung von Zeichen

Mit dem Zeichen-Ersetzungs-Assistenten können Projekte erstellt werden, die eine Reihe von Buchstaben durch jeweils andere Schriftzeichen oder Literals ersetzen. Das kann z.B. bei der Konvertierung von Text-Dateien nötig sein, die auf einem anderen Betriebssystem geschrieben wurden.

Ein Zeichen-Ersetzungs-Projekt ist einem Wort-Ersetzungs-Projekt sehr ähnlich. Da es jedoch nur

eine begrenzte Menge an Zeichen gibt, braucht die Ersetzungsliste nicht eingetippt zu werden, sondern man kann in einer vollständigen Tabelle für jedes Zeichen ein anderes aus einer Liste auswählen. Zeichen, für die keine Ersetzung ausgewählt wird, werden unverändert in den Zieltext kopiert. Es ist auch möglich, statt eines Ersetzungszeichens aus der rechten Box einen literalen Ersetzungsausdruck direkt in das entsprechende Feld der Tabelle zu schreiben. Die Tabelle geht dabei in den Edier-Modus über.

Wählen sie **zuerst** ein Zeichen aus der Tabelle aus. Sobald sie ein Zeichen in der rechten Box auswählen, wird der Ersetzungs-Text der aktuellen Tabellenspalte überschrieben.



Der Inhalt der Spalte: "Ersetzen durch ...", kann abgespeichert und später neu geladen werden.

Wenn in allen Zeilen der Tabelle die gewünschten Zuordnungen bestehen und sie den Bestätigen-Schalter betätigt haben, durch den der Edier-Modus verlassen wird, bekommen sie auf der nächsten Seite den Text einer Produktion angezeigt. Zeichenersetzungs-Projekte bestehen aus nicht mehr als nur dieser einfachen Produktion. Z.B. sieht die Produktion für die Konvertierung eines Atari-Textes wie folgt aus:

```
(
    "□"     {{ out << xState.str(-1) << "";}}
  | "□"     {{ out << xState.str(-1) << "";}}
  | "□"     {{ out << xState.str(-1) << "";}}
  | "□"     {{ out << xState.str(-1) << "";}}
  | "□"     {{ out << xState.str(-1) << "";}}
  | "□"     {{ out << xState.str(-1) << "";}}
  | "□"     {{ out << xState.str(-1) << "ü";}}
  | "„"     {{ out << xState.str(-1) << "ä";}}
  | "Ž"     {{ out << xState.str(-1) << "Ä";}}
  | """     {{ out << xState.str(-1) << "ö";}}
  | "™"     {{ out << xState.str(-1) << "Ö";}}
  | "š"     {{ out << xState.str(-1) << "Ü";}}
  | "œ"     {{ out << xState.str(-1) << "§";}}
  | "ž"     {{ out << xState.str(-1) << "ß";}}
  | SKIP     {{ out << xState.copy(); }}
)*
```

Wenn sie nun den Assistenten beenden, ist das Projekt fertig für die Anwendung.

Im Verzeichnis:

..\TextTransformer\Beispiele\Assistenten

befinden sich auch zwei Zeichenlisten:

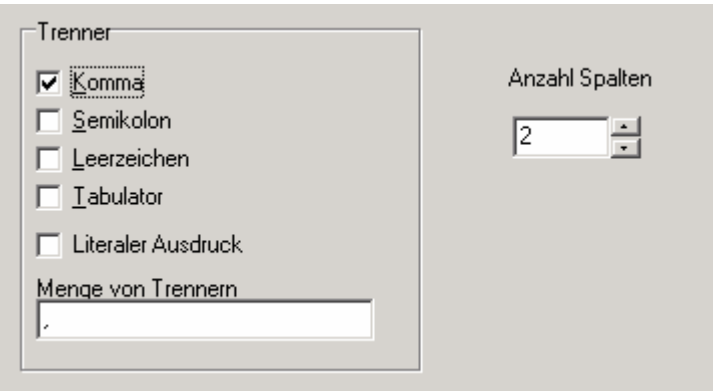
ANSI2DOS.txt und DOS2ANSI.txt

mit denen Projekte zur Konverierung zwischen dem ANSI- und dem DOS-Zeichsatz erzeugt werden können.

9.2.9.2.1.4 CSV-Assistent

Mit dem Kürzel *CSV* (Character Separated Values) werden Dateien benannt, deren Zeilen aus Daten bestehen, die durch Kommata oder andere Trennzeichen voneinander getrennt sind. Viele Datenbankanwendungen können solche Dateien lesen und schreiben.

Der hier beschriebene Assistent erlaubt es, die einzelnen Daten zu extrahieren, um sie dann zu verändern oder anders anzuordnen. Dabei wird davon ausgegangen, dass das Trennzeichen innerhalb der Daten nicht vorkommt. Sollte dies bei ihrer CSV-Datei nicht der Fall sein, so können sie den Assistenten verwenden, der aus einem Beispielstext einen Zeilenparser generiert.



Als Zeilentrenner kann nicht nur ein Komma, sonder es können auch beliebige andere Zeichen angegeben werden. Es ist auch möglich mehrere Trenner anzugeben, wobei jedoch jeweils stets nur einer aus dieser Menge zwischen zwei Feldern stehen darf. Sind die Spalten durch mehr als ein Zeichen voneinander getrennt, so kann man den Trenner auch als einen literalen Ausdruck definieren.

Nachdem die Anzahl der Spalten eingestellt wurde, kann auf die nächste Seite des Assistenten gewechselt werden, wo die gewünschte Art von Aktionen ausgewählt werden kann. Auf der dann folgenden Seite ist dann der Text einer Produktion zu sehen ist, die aus den Einstellungen generiert wird.

Wenn ein durch ein Komma getrennter zwei spaltiger Text in String-Variablen geschrieben wird, sieht die Produktion so aus:

```
{{
str sCol1, sCol2;
}}
```

```
(
SKIP     {{sCol1 = xState.str(); }}
","
SKIP     {{sCol2 = xState.str(); }}
EOL
)

{{
// out << Hier können Sie die Spalten in der gewünschten Form ausgeben.
}}
```

Der Kommentar in der vorletzten Zeile könnte nun z.B. ersetzt werden durch:

```
out << sCol2 << "," << sCol1 << endl;
```

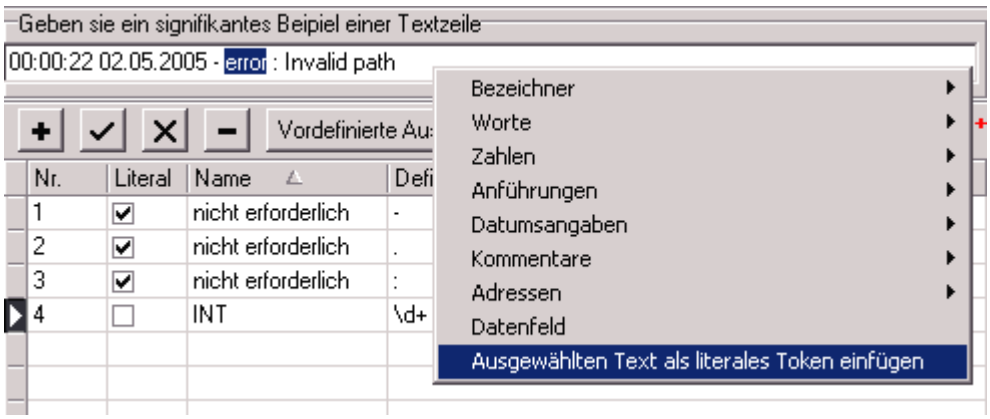
Damit würde ein Ausgabetext erzeugt, in dem die Spalten des Quelltextes vertauscht wären.

Ein (englisches) Video zu diesem Assistenten gibt es unter:

http://www.texttransformer.com/Videos_en.html

9.2.9.2.1.5 Zeilenparser aus Beispielstext erzeugen

Besteht eine Datei aus Zeilen, die jeweils die gleiche Struktur haben - z.B. eine Log-Datei -, so kann man mit dem hier beschriebenen Assistenten einen vollständigen Parser für diese Datei erzeugen.



Zunächst kopiert man eine typische Zeile in das Feld über der Tabelle. In die Tabelle müssen dann die Definitionen der Token eingegeben werden, mit denen die Zeile analysiert werden soll. Diese Eingabe ist besonders einfach für literale Ausdrücke. Im Popup-Menü, das nach Klick mit der rechten Maustaste erscheint, gibt es den Menü-Punkt: Ausgewählten Text als literales Token einfügen. Mit dieser Funktion kann ein Text, der in dem Edit-Feld mit der Maus markiert wurde, direkt als Token in die Tabelle übernommen werden.
Sonst geschieht die Eingabe der Werte in die Tabelle genauso wie bei den Tabellen anderer Assistenten.

Wird auf der nächsten Seite des Assistenten für die Aktionen die direkte Ausgabe gewählt, so erhält

man nach Beendigung des Assistenten folgende Chapter-Produktion:

```
INT     {{out << xState.copy(); }}
":"     {{out << xState.copy(); }}
INT     {{out << xState.copy(); }}
":"     {{out << xState.copy(); }}
INT     {{out << xState.copy(); }}
INT     {{out << xState.copy(); }}
"."     {{out << xState.copy(); }}
INT     {{out << xState.copy(); }}
"."     {{out << xState.copy(); }}
INT     {{out << xState.copy(); }}
"-"     {{out << xState.copy(); }}
"error"     {{out << xState.copy(); }}
":"     {{out << xState.copy(); }}
SKIP     {{out << xState.copy(); }}
EOL
```

Diese Produktion besteht aus den Token der Tabelle und zwar in der Reihenfolge, wie sie im Beispielstext aufgefunden werden. Text-Teile, die von den Token der Tabelle nicht erkannt werden, werden mit dem SKIP-Token erfasst.

9.2.9.2.1.6 Kopf/Kapitel/Fuß

Eine häufige allgemeine Struktur von Texten ist die Unterteilung in einen Text-Kopf, einige Kapitel und einen Text-Fuß. In einem Buch z.B. wäre das Inhaltsverzeichnis der Kopf-Teil und das Sachverzeichnis der Fuß-Teil. Der hier beschriebene Assistent erzeugt einen Rahmen für eine solche Struktur. Für Kopf, Kapitel und Fuß werden jeweils eigene Produktionen erzeugt, die von der Startregel aufgerufen werden.

In die **Kopf (Header)-** und die **Fuß (Footer) -Produktion** wird zunächst nur das **SKIP**-Symbol eingesetzt. Die exakte Struktur dieser Teile muss von Hand geschrieben werden.

Für die **Kapitel (Chapter)** -Produktion ist es möglich, einige aufeinander folgende Token bereits im Assistenten zu definieren.

Die Eingabe der Werte in die Tabelle geschieht genauso wie bei den Tabellen anderer Assistenten.

Wenn das Token, mit dem die Kapitel-Produktion beginnt, eindeutig so spezifiziert ist, dass es nicht im Kopf-Teil vorkommen kann, so können mit dem vom Assistenten erstellten Rahmen bereits ausführbare Parser erzeugt sein.

Beispiel

Um die Schlüsselworte in einer HTML-Datei zu bearbeiten, kann man sich über signifikante Ausdrücke in der Datei bis zur gesuchten Stelle "durchhangeln".

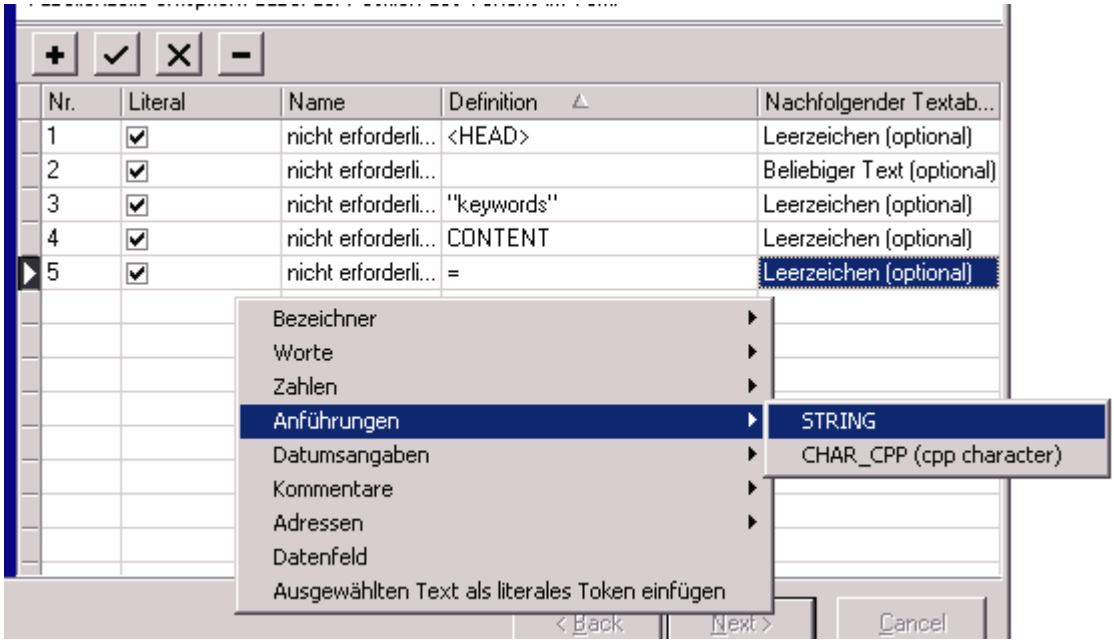
Der verkürzte Anfang der Datei:

C:\TextTransformer\Beispiele\Assistenten\textkonverter.html

sieht wie folgt aus:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<HTML>
<HEAD>
  <meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
  <meta name="GENERATOR" content="TextTransformer">

  <META NAME="keywords" CONTENT="Text Konverter">
</HEAD>
...
```

Wenn eine Reihe von Token so definiert werden, wie im Bild dargestellt:



wird durch die **Header**-Produktion der Text bis zum Token "<HEAD>" übersprungen, dann wird der Ausdruck "\"keywords\"" gesucht, dann "CONTENT" und "=" und schließlich der gesuchte string gefunden. Mit der **Footer**-Produktion wird dann der Rest des Textes übersprungen.

Wird auf der nächsten Seite des Assistenten für die Aktionen die direkte Ausgabe gewählt, so erhält man nach Beendigung des Assistenten folgende Chapter-Produktion:

```
"<HEAD>"     {{out << xState.copy(); }}
( SKIP     {{out << xState.copy(); }} )?
"CONTENT"     {{out << xState.copy(); }}
"="     {{out << xState.copy(); }}
STRING     {{out << xState.copy(); }}
```

Hier kann nun leicht die mit *STRING* verbundene Aktion den eigenen Wünschen gemäß geändert werden.

9.2.9.2.1.7 Aktionen

Für die vom Assistenten erzeugten Parser können automatisch Aktionen generiert werden, die zumeist darin bestehen, den jeweils erkannten Text zu kopieren. Diese Aktionen können dann mit wenig Arbeit so modifiziert werden, dass der Ausgabetext die gewünschte Form erhält.

Direkte Ausgabe

Für jedes Token wird eine Aktion erzeugt, die den erkannten Text-Abschnitt samt den ausgelassenen Zeichen direkt in die Ausgabe schreibt. Z.B.:

```
Token1 {{ out << xState.copy();}}
Token2 {{ out << xState.copy();}}
```

Das direkte Schreiben in die Ausgabe ist die effizienteste Art den Text zu verarbeiten und sollte daher, wenn möglich, gewählt werden. Die Reihenfolge der Text-Abschnitte bleibt dabei allerdings unverändert.

In String-Variable schreiben

Für jedes Token wird eine Aktion erzeugt, die den erkannten Text-Abschnitt samt den ausgelassenen Zeichen in eine String-Variable schreibt. Z.B.:

```
Token1 {{ s1 = xState.copy();}}
Token2 {{ s2 = xState.copy();}}
```

Hier werden die erkannten Text-Abschnitte dupliziert, können dann aber in beliebiger Reihenfolge ausgegeben werden. Z.B.:

```
out << s2 << s1;
```

Parse-Baum erzeugen

Für jedes Token wird eine Aktion erzeugt, die den erkannten Text-Abschnitt samt den ausgelassenen Zeichen in eine Knoten-Variable schreibt. Z.B.:

```
Token1 {{ nRule.add("Token1", xState.copy());}}
Token2 {{ nRule.add("Token2", xState.copy());}}
```

Ein Parse-Baum erlaubt eine vielseitige und mehrfache Weiterverarbeitung seiner Knoten. Jedoch ist es nicht trivial korrekte Routinen für diese Verarbeitung zu schreiben.

DOM erzeugen

Mit dieser Option wird wie oben ein Parse-Baum erzeugt, aber aus dnode's statt aus node's. Damit wird ein XML-Dokument erzeugt, das schließlich in die Ausgabe geschrieben wird.

Keine Aktionen

Wenn nur ein kleiner Teil des erkannten Textes ausgegeben werden soll, oder wenn obige Methoden miteinander kombiniert werden sollen, so empfiehlt es sich, den Code für die Aktionen von Hand zu schreiben.

9.2.9.2.2 Eine Produktion aus einem Beispielstext erzeugen

Mit diesem Assistenten können sie aus einem Beispielstext eine Produktion erzeugen, die aus einer einfachen Abfolge von Token besteht. Die Abfolge ergibt sich aus der Reihenfolge, in der die Token im Text gefunden werden.

Dieser Assistent ist dem Assiostenten sehr ähnlich, der einen Zeilenparser aus einem Beispielstext erzeugt.

Es gibt zwei Unterschiede:

1.Der Beispielstext muss nicht aus genau einer Zeile bestehen. Wenn ein Teil der Quelltextes im Eingabefenster selektiert ist, bevor der Assistent geöffnet wird, wird dieser Teil automatisch als Beispiel genommen.

2. Token, die im aktuellen Projekt bereits definiert sind, werden automatisch in die Tabelle der Token eingefügt, die zur Analyse des Beispieltextes verwendet werden. Token, die nicht getestet werden sollen, können aus der Tabelle gelöscht werden.

9.2.9.2.3 Parameter-Assistent

Der Parameter-Assistent Assistent vereinfacht die Erzeugung eines einheitlichen Parameters oder einer einheitlichen Variablen-Deklaration für mehrere Skripte.

Auf den einzelnen Seiten des Assistenten gibt es jeweils eine kurze Erläuterung, zu den angebotenen Optionen. Hier wird daher nur am Beispiel einer einzelnen Produktion kurz skizziert, was mit dem Assistenten als Resultat für alle Produktionen und Token erreicht werden kann. Die Produktion:

```
A ( ) ::= _a ( A | B )
```

kann im umfassendsten Fall mit einem Parameter *xParam* vom Typ *type* und mit einer lokalen Deklaration einer solchen Parameter-Variablen ausgestattet werden.

```
A ( type& xParam ) ::=

{{
type  Param;
}}
_a[Param]
(
     A[Param]
   | B[Param]
)
```

Wenn die Option zur Erzeugung von Deklarationen nicht gesetzt wird erhält man:

```
A ( type& xParam ) ::=

_a[xParam]
(
     A[xParam]
   | B[xParam]
)
```

Wenn xParam z.B. vom Typ *str* ist, können diese Gerüste dann leicht so ergänzt werden, dass die Referenz-Variable xParam nach Durchlaufen des gesamten Parsers den gewünschten Zieltext enthält. Für das benannte Literal _a könnte die Ergänzung so aussehen:

```
_a( str& xParam ) ::=

{{
   xParam += xState.copy();
}}
```

Würden für alle Token entsprechende Aktionen definiert, so enthielte xParam nach Abarbeitung des Parsers eine Kopie des Quelltextes.

Die automatische Erzeugung des Codes - im Beispiel `"[xParam]"` - der Übergabe des Parameters an die aufgerufenen Produktionen und Token ist nur möglich, wenn die Option: "für alle Produktionen und Token" gewählt wird.

Der Baum-Assistent funktioniert ganz ähnlich wie der Parameter-Assistent für den speziellen Typ node.

9.2.9.2.4 Baum-Assistent

Der Baum-Assistent vereinfacht die Erzeugung von Baum-Knoten für mehrere Skripte. Im Unterschied zum Parameter-Assistenten kann der Baum-Assistent auch Code erzeugen, der die Knoten-Parameter in einen Gesamt-Baum einfügt, Für vollständige Projekte kann der Assistent auch Aktionen zur Erzeugung von Baumknoten für die rein literalen Token einfügen und er kann die jeweils neuen Knoten als Aufrufparameter an die untergeordneten Produktionen und Token weitergeben. All dies dient letztlich der Generierung von Parse-Bäumen und ihrer Auswertung mittels Funktionstabellen. Auch für die letzteren existiert ein Assistent. In der Hilfe zum Funktions-Tabellen-Assistenten gibt es weiter Erläuterungen zur Benutzung von Parse-Bäumen

Wählt man die Option, Knoten innerhalb der Produktionen erzeugen zu lassen, so sieht für die einzelne Produktion:

```
A ( ) ::= (_a | "b" ) ( A | B )
```

das Resultat, das erzielt wird, wenn man die Einstellung "für alle Produktionen und Token" wählt, folgendermaßen aus:

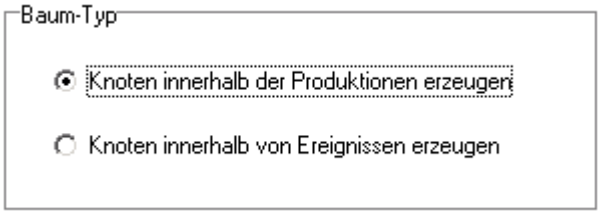
```
A ( node& xn ) ::=

{{
node  n("A");
```

```
xn.addChildLast(n);
}}
(
   _a[n]
  | "b"   {{n.add( "LITERAL", State.str() );}}
)
(
    A[n]
  | B[n]
)
```

Die automatische Erzeugung des Codes - im Beispiel "[n]" - der Übergabe des Knotens an die aufgerufenen Produktionen und Token ist nur möglich, wenn die Option: "für alle Produktionen und Token" oder "komplett" gewählt wird.

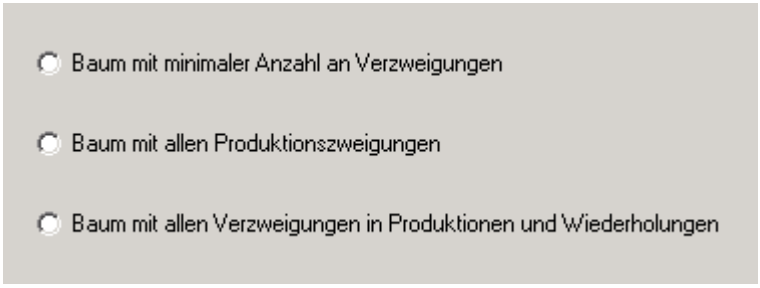
9.2.9.2.4.1 Baum-Typ

Wenn man den *Baum-Assistenten* im Hilfe-Menü aufruft, so erscheint zunächst eine Auswahlmöglichkeit für die Art und Weise, wie der Baum erzeugt werden soll.



Im Java-Beispiel ist die Bedienung des Assistenten für die erste Option ausführlich vorgestellt. Der Vorteil dieser Option ist, dass durch Änderungen des eingefügten Codes, die Baumerzeugung leicht gesteuert werden kann. So können z.B. gezielt bestimmte Elemente ignoriert werden.

Der Vorteil der Baumerzeugung in den Ereignissen dagegen ist, dass die Baumerzeugung vom Code der Produktionen unabhängig geschieht.
Es gibt drei Arten von Bäumen, die sich so generieren lassen:



Sie unterscheiden sich zum einen durch die Anzahl der in ihnen dargestellten Elemente und zum anderen durch den Zeitpunkt der Erzeugung. Der Baum mit minimaler Anzahl an Verzweigungen, wird aus einem Hilfscontainer erst produziert, nachdem die letzte der in der Startproduktion aufgerufenen Produktionen verlassen ist. Die anderen Bäume werden während des Parsens erzeugt und können bereits zu diesem Zeitpunkt benutzt werden.

Ein Video (englisch) , dass diesen Bamumtyp demonstriert gibt es unter:

http://www.texttransformer.com/Videos_en.html

9.2.9.2.5 Funktions-Tabellen-Assistent

Den Funktions-Tabellen-Assistenten gibt es in einer kleinen und in einer ausführlichen Version. Der kleine **Schnellassistent** zur Erweiterung einer bestehenden Funktionstabelle durch einzelne Funktionen erscheint, wenn mit der rechten Maustaste auf den Namen einer Funktions-Tabelle in der Liste der Klassenelemente geklickt wird. Der **ausführliche Assistent**, der hier beschrieben wird, kann entweder vom Schnellassistenten aus aufgerufen werden oder aber als Eintrag des Hilfe-Menüs aufzurufen.

Mittels des Funktions-Tabellen-Assistenten können neue Tabellen erstellt und vorhandene Tabellen erweitert werden, wobei es möglich ist ganze Gruppen von Zuordnungen auf einmal zu erstellen. Im Zusammenspiel mit dem Baum-Assistenten kann sogar für ein vollständiges Projekt automatisch der Code zur Erzeugung eines kompletten Parse-Baums in sämtliche Produktionen eingefügt werden. Diese Möglichkeit sollte für ein Projekt nur einmal ausgeführt werden, da eine Wiederholung zu Namenskonflikten führt.

Die Namen der erzeugten Funktionen und der Funktionstabelle werden aus einem Basisnamen generiert:

Wird "m_ft" am Anfang des Namens der Funktionstabelle im nachhinein geändert wird, so arbeitet der Assistent nicht mehr korrekt mit der Tabelle zusammen.

Hintergrund:

Eines der Paradigmen zur Erzeugung von Transformationsprogrammen ist es, in einem ersten Schritt einen Parse-Baum zu erzeugen, der dann auf vielfältige Weise zur Generierung von Ausgaben verwendet werden kann. In dem Parse-Baum werden Produktionen und Token durch Baumknoten repräsentiert.
Ein empfehlenswertes Schema im TextTransformer solche Bäume zu konstruieren wird durch den Funktions-Tabellen-Assistenten unterstützt. Nach diesem Schema, werden die Namen der jeweiligen Produktionen bzw. Token als Labels der Knoten verwendet. Im TextTransformer können nun spezielle Tabellen angelegt werden: die Funktionstabellen, die den Labels der Knoten Funktionen zuordnen. Diese Zuordnung ist damit indirekt zugleich eine Zuordnung von Funktionen zu den Produktionen bzw. Token. Jede dieser Funktionen dient der Verarbeitung des zugehörigen Knotens und damit der Verarbeitung der Produktion bzw. des Tokens.

Häufig werden verschiedene Token oder Produktionen auf die gleiche Art und Weise zu behandeln sein. Z.B. sind viele der durch Token erkannten Texte unverändert wieder auszugeben. Daher können verschiedene Token oder Produktionen mit der gleichen Funktion verarbeitet werden. Erfordert hingegen ein spezielles Token eine spezielle Behandlung, so wird für seinem Knoten eine spezielle

Funktion geschrieben, die nur ihm und keinem anderen Token zugeordnet wird.

Beispiel:

Name des Skripts	Label des Knotens	Funktion
normal1	normal1	Handle_Default
normal2	normal2	Handle_Default
special1a	special1a	Handle_special1
special1b	special1b	Handle_special1
special2	special2	Handle_special2

Die Initialisierung der Funktionstabelle *m_ft* sieht dann so aus:

```
m_ft.add("", "Handle_Default");
m_ft.add("special1a", "Handle_special1");
m_ft.add("special1b", "Handle_special1");
m_ft.add("special2", "Handle_special2");
```

In den Skripten müssen die entsprechenden Knoten erzeugt und dem gesamten Parse-Baum hinzugefügt werden. Eine Referenz auf den letzten Knoten des Parse-Baum wird als Parameter an das Skript übergeben:

```
Parameter:
        node& xNode
Text:
        node n("normal1");
        xNode.addChildLast(n);
```

oder

```
Parameter:
        node& xNode
Text:
        node n("special2");
        xNode.addChildLast(n);
```

Ganz entsprechend für die anderen Skripte.
Beim Aufruf der Skripte muss nun jeweils der zusätzlicher node-Parameter übergeben werden:

```
normal1[n]
special2[n]
```

9.2.9.2.5.1 Schnellassistent für Funktions-Tabellen

Der Funktions-Tabellen-Assistent erleichtert die Erweiterung einer Funktionstabelle mit neuen Funktionen. Der Assistent erscheint, wenn mit der rechten Maustaste auf den Namen einer Funktions-Tabelle in der Liste der Klassenelemente geklickt wird.

In die oberen beiden Felder sind das Label eines zu behandelnden Knotentyps, und der Name der Funktion, die diesen Knoten behandeln soll einzugeben. Wird dann der ***Einfügen***-Schalter betätigt, so wird die aktuelle Funktions-Tabelle um einen Eintrag erweitert:

```
m_ftExpr.add("chars", "Handle_chars");
```

und es wird zugleich eine neue Funktion erzeugt, die die gleichen Parameter hat, wie für die Funktions-Tabelle definiert und deren Rückgabetyp dem Typ der Funktions-Tabelle entspricht.
In der Auswahlbox ***Art*** kann für die neue Funktion zuvor ein Gerüst-Typ gewählt werden.
Wenn z..B. die aktuelle Funktions-Tabelle vom Typ *str_mstrfun* ist, mit einem node-Parameter, so wird mit dem ***iteration***-Gerüst folgende Funktion auf der Element-Seite eingefügt:

Name: Handle_chars
Typ: str
Parameter: const node& xnNode
Text:

```
{{
str s;
node pos = xnNode.firstChild();
while(pos != node::npos)
{
  s += m_ftExpr.visit(xState, pos);
  pos = pos.nextSibling();
}

return s;
}}
```

Für den *Art* ***value*** wird der folgende Text erzeugt:

```
{{
 return xnNode.value();
}}
```

Für die *Art* ***empty*** wird nur die Klammer erzeugt.

Das Funktionsgerüst kann dann von Hand weiter bearbeitet werden.

Eine einfache Methode um eine **konsistente Verlinkung** herzustellen zwischen

1. einer Funktions-Tabelle
2. einer zu ihr passenden Funktion
3. den Knoten, die eine Produktion repräsentieren (bzw. den durch sie erkannten Text)

ist, das Label des Knotens nach der Produktion zu nennen. Daher enthält der Assistent eine Liste der **Namen der Produktionen**, zur Auswahl **als Label**.
Wenn dei **Check-Box *Baumknoten in die Produktion einfügen*** aktiviet ist,
wird den Parametern der Produktion ein zusätzlicher Knoten-Parameter vorangestellt, und an den Anfang des Textes der Produktion wird folgender Code geschrieben:

```
node n("chars");
xNode.addChildLast(n);
```

Für die Aufrufe der Produktion muss von Hand der zusätzliche Parameter eingefügt werden. Wenn in der aufrufende Funktion ebenfalls der durch den Assistenten geschriebene Code steht, ist dieser parameter in der Regel: n.

9.2.9.2.6 Eingabe-Tabellen

In mehreren Assistenten gibt es Tabellen zur Eingabe von Ausdrücken. Die Bedienung dieser Tabellen ist jeweils gleich.

In der Werkzeugleiste befinden sich vier Schalter:

 Neue Zeile einfügen

 Wert des edierten Feldes übernehmen

 Veränderungen im edierten Feld verwerfen.

 Zeile entfernen

Edier-Modus

Um ein schnelles Arbeiten in der Tabelle zu ermöglichen, wird der Assistent in einen Editiermodus versetzt, in dem einige Tastatur-Eingaben eine andere Funktion haben als üblich. Im Edier-Modus können sie direkt in die Felder schreiben. Die Eingabe kann durch den Bestätigen-Schalter oder die Eingabetaste bestätigt werden. Mit den Pfeiltasten können sie dann in der Tabelle navigieren, mit **Strg+Eing.**.eine neue Zeile anfügen.

Der Edier-Modus wird mit dem Bestätigen, Abbrechen oder Entfernen-Schalter verlassen oder durch einen **einzelnen** Druck auf die Esc-Taste. Ein zweiter Druck würde den Assistenten schließen. Achtung: auch die Eigabetaste erhält nach dem verlassen des Editier-Modus seine alte Bedeutung

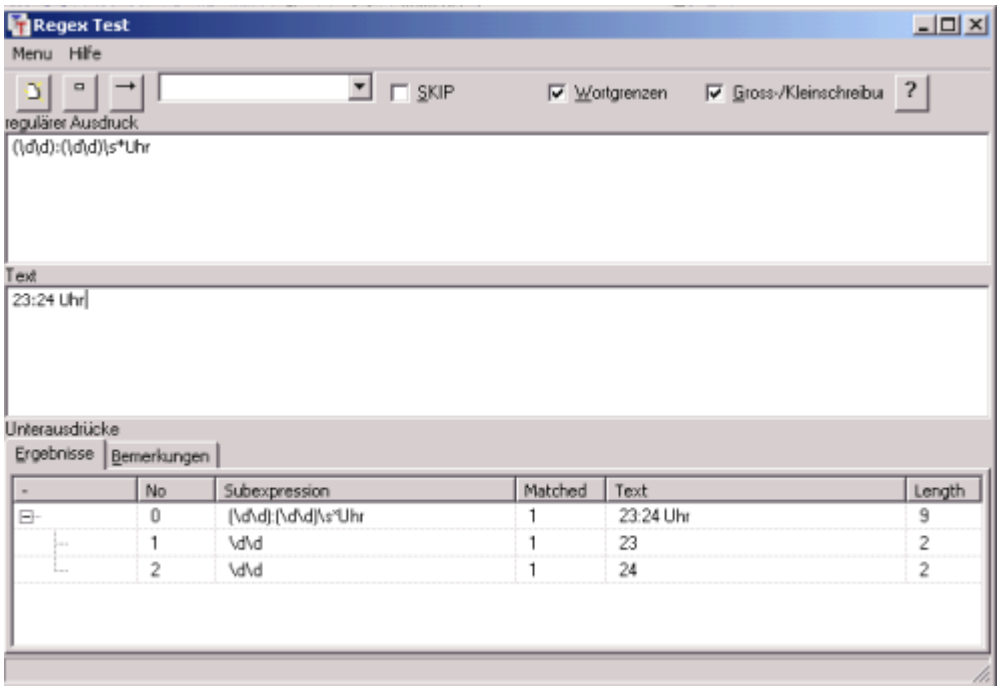
zurück: sie aktiviert die Funktion des aktuellen Kontroll-Elements.

Literal

In einigen Tabellen gibt es eine Spalte mit der Überschrift "Literal". Wird das Kästchen in dieser Spalte aktiviert, so braucht für das Token keine Name vergeben zu werden und der Definitionstext wird in literaler Bedeutung genommen. D.h., die Zeichen, die in regulären Ausdrücken eine Metabedeutung haben, haben diese hier nicht: **jeder Buchstabe bedeutet sich selbst**. Auch dem Anführungszeichen und dem Backslash muss hier nicht mit ein (zweiter) Backslash vorangestellt werden, wie bei der Definition literaler Token direkt innerhalb einer Produktion.

9.2.9.3 Regex Test

Unter dem Menüeintrag *Hilfe* gibt es den Unterpunkt *Regex Test*. Hier lässt sich ein Dialog aufrufen, in dem reguläre Ausdrücke einzeln getestet werden können. Die Ausdrücke sind hier in der gleichen Syntax anzugeben, wie auf der Tokenseite. D.h. vor den Metazeichen regulärer Ausdrücke muss ein Backslash '\' gesetzt werden, wenn das Zeichen in literaler Bedeutung interpretiert werden soll.



Die Dialogbox besteht aus einem Menü, einer Werkzeugleiste und drei Unterfenstern.

Die Menüpunkte sind auch direkt über die Schalter der Werkzeugleiste ausführbar:

Edit-felder löschen

Die Texte in den drei Unterfenstern der Dialogbox werden gelöscht.

Kompilieren

Der reguläre Ausdruck im obersten Fenster wird kompiliert. Je nachdem, ob die Syntax des Ausdrucks korrekt ist oder nicht wird der Erfolg bestätigt, oder das Auftreten eines Fehlers angezeigt.

Ausführen

Der reguläre Ausdruck im obersten Fenster wird kompiliert und auf den Text im mittleren Fenster angewandt. War der reguläre Ausdruck syntaktisch korrekt, wird er samt seinen Unterausdrücken im untersten Fenster aufgelistet. Rechts neben den jeweiligen Unterausdrücken wird angegeben, welcher Teil des Textes durch ihn erkannt wurde. (Im abgebildeten **Beispiel** enthält der 1. Unterausdruck die Stunden- und der zweite die Minutenzahl der Uhrzeit. Ein weiteres Beispiel wird im Waechterprojekt gezeigt.)

Auswahlbox

In der Auswahlbox der Werkzeugleiste sind alle Token des aktuellen Projekts aufgelistet. Bei der Wahl eines dieser Token, wird sein Definitionstext in das oberste Fenster übernommen.

Optionen

SKIP

Die Option SKIP ist normalerweise deaktiviert. Eine Übereinstimmung mit dem Text wird dann nur gefunden, wenn sie am Textanfang liegt, so wie beim beim Parsen eines Textes das nächste Token zumeist auf die aktuelle Textposition passen soll.
Ist die Option SKIP eingeschaltet, so wird nach der nächsten Stelle im gesamten Texte gesucht, auf die der reguläre Ausdruck passt. Wird eine **Übereinstimmung am Textanfang** gefunden, wird dies **als Fehler gewertet**.

Wortgrenzen

Die Wortgrenzen-Option hier wirkt sich für Literale auf die gleiche Weise aus, wie die Wortgrenzen-Option in den Projekteinstellungen. Die Auswertung geschieht hier wie bei den von TETRA erzeugten Parsern durch einen speziellen ternären Baum. **Bei regulären Ausdrücken hat diese Option hier keine Auswirkungen**, da der Ausdruck fertig aus dem obersten Editfeld übernommen wird.
Die SKIP-Ausdrücke einer Produktion werden hingegen erst beim Kompilieren des Projekts erzeugt und können somit literale Unterausdrücke mit Wortgrenzen enthalten. Wortgrenzen werden dann durch Einfügung der Anker "\<" und "\>" formuliert.

Gross-/Kleinschreibung

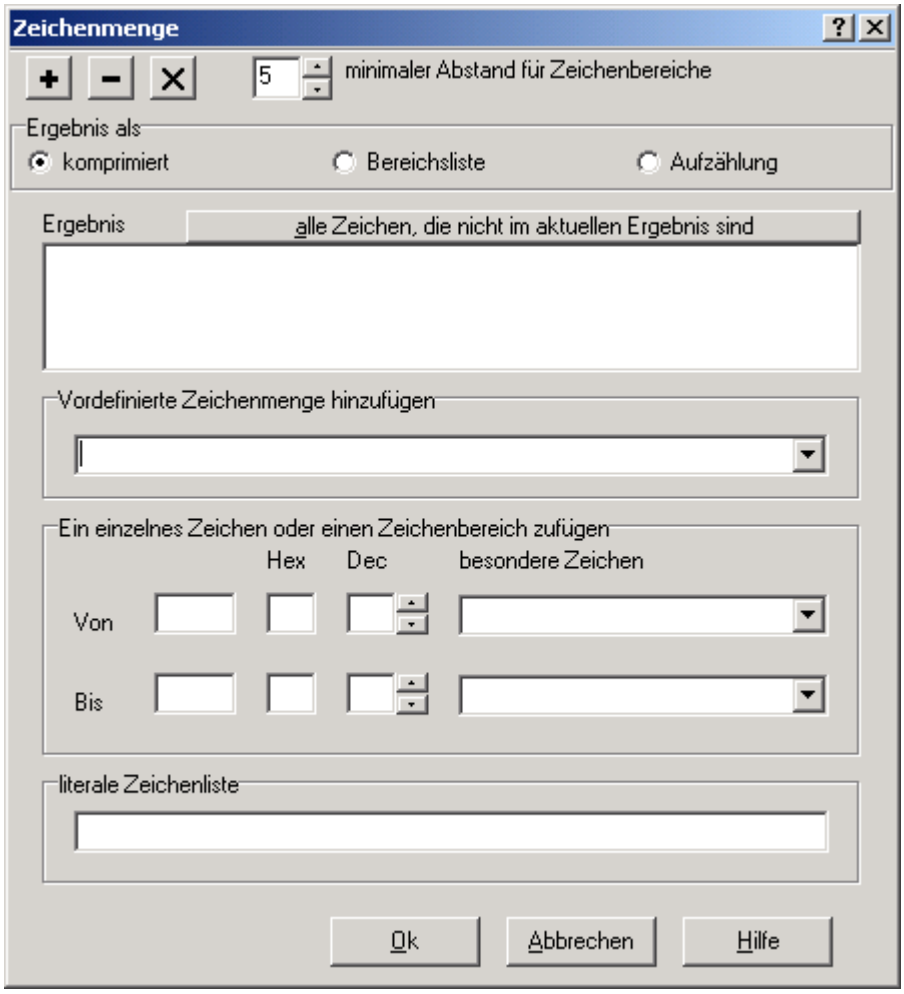
Die *Gross-/Kleinschreibung*-Option hier wirkt sich auf die gleiche Weise aus, wie die *Gross-/Kleinschreibung-Option* in den Projekteinstellungen.

Anmerkung:

Der Test der regulären Ausdrücke arbeitet in gleicher Weise, wie die Scanner in einem TextTransformer Projekt. Ist der Ausdruck im obersten Fenster ein **Literal**, so wird dies im Ergebnisfenster angegeben. Der Ausdruck wird dann auch nicht als regulärer Ausdruck, sondern eben als Literal getestet. Während "" und '\"' für den regulären Ausdruck ununterschieden sind, gilt dies nicht für Literale.

9.2.9.4 Zeichenmengen-Berechner

Der Zeichenmengen-Berechner dient dazu Zeichenmengen schrittweise zu erstellen, indem der jeweiligen Ergebnis-Menge weitere Zeichen hinzugefügt werden oder bestimmte Zeichen entfernt werden.



Die hinzuzufügenden oder zu entfernenden Zeichen können aus einer Liste vordefinierte Zeichenmengen ausgewählt werden, oder es kann ein Zeichenbereich angegeben werden, oder es kann eine Liste von Zeichen oder ein einzelnes Zeichen im eingegeben werden

Für die literale Zeichenliste gelten die gleichen Regeln wie für die Zeichen eines Strings. D.h. es können z.B. auch Zeilenumbrüche '\n' etc. eingegeben werden. Daher muss auch das Backslashzeichen verdoppelt werden: \\, um es einer Liste hinzuzufügen.

In der Werkzeug-Leiste des Zeichenmengen-Berechners dient der mit '+' gekennzeichnete Schalter dazu,

Zeichen zur Ergebnis-Zeichenmenge hinzuzufügen. Die hinzuzufügenden Zeichen müssen zuvor mittels der darunterliegenden Dialog-Elemente ausgewählt worden sein.

Der mit '-' gekennzeichnete Schalter

erlaubt es Zeichen aus der Ergebnis-Zeichenmenge zu entfernen. Die zu entfernenden Zeichen müssen zuvor mittels der darunterliegenden Dialog-Elemente ausgewählt worden sein. Werden Zeichen ausgewählt, die in der Ergebnismenge nicht vorhanden sind, so hat dies keine Auswirkung.

Der mit 'x' gekennzeichnete Schalter

löscht die bisherige Auswahl.

Dargestellt werden kann das **Ergebnis als**

komprimierte Darstellung

Zeichen werden in Zeichenmengen zusammengefasst, sofern die Menge vollständig ist. Die restlichen Zeichen werden als Bereichslisten dargestellt.

Bereichsliste

In der ANSI-Tabelle aufeinander folgende Zeichen werden zu Bereichen zusammengefasst. Z.B. wird aus 1,2,3,4,5,6 der Bereich 1-6. Die restlichen Zeichen werden einzeln aufgezählt.
Ob die Zeichen zu einem Bereich zusammengefasst werden oder nicht, hängt auch von der folgenden Einstellung ab.

minimaler Abstand für Zeichenbereiche

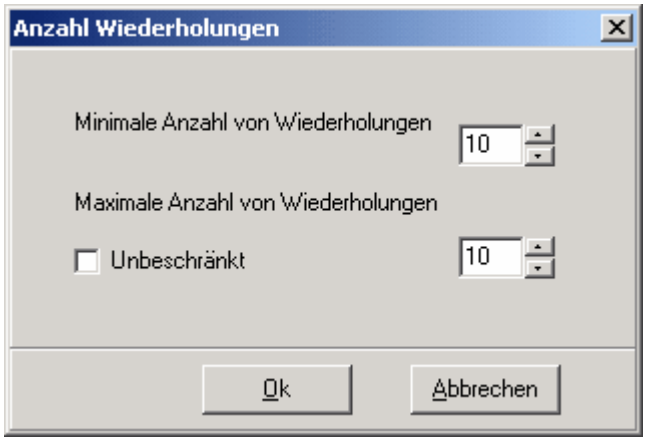
Zeichen werden nur dann zu einem Bereich zusammengefasst, wenn gemäß der ANSI-Liste zwischen dem kleinsten und dem größten Zeichen mindestens soviele weitere Zeichen liegen, wie durch den minimalen Abstand für Zeichenbereiche gefordert ist.

Aufzählung

Sämtliche Zeichen werden einzeln aufgezählt.

Mit dem länglichen Schalter oberhalb der Ergebnis-Liste: ***alle Zeichen, die nicht im aktuellen***

Ergebnis sind, kann die bisherige Auswahl invertiert werden, d,h, alle Zeichen werden aufgelistet, die zuvor nicht ausgewählt waren.



Wenn der Dialog mit *Ok* verlassen wird, erscheint ein zweiter Dialog, in dem eingegeben werden kann, wie viele Zeichen der ausgewählten Menge aufeinander folgen sollen. Nach nochmaligem *Ok* wird der resultierende Ausdruck in die Zwischenablage kopiert.

Beispiel:

In der englischen Dokumentation zur Wikipedia:

http://en.wikipedia.org/wiki/Wikipedia:How_to_edit_a_page

heißt es:

In the URL, all symbols must be among:
A-Z a-z 0-9 . _ \ / ~ % - + & # ? ! = ( ) @ \x80-\xFF

Um diese Zeichenklasse zu konstruieren, können folgende Schritte ausgeführt werden:

1. die Zeichenklasse: "die alphanumerischen Zeichen und der Unterstrich" hinzufügen.

2. den Zeichenbereich von Hex 80 bis Hex ff hinzufügen

3. die Zeichenliste: ".\\/~%+-&#?!=()@" hinzufügen.

Achtung: die Liste daf keine Leerzeichen enthalten und der Backslash muss verdoppelt werden.

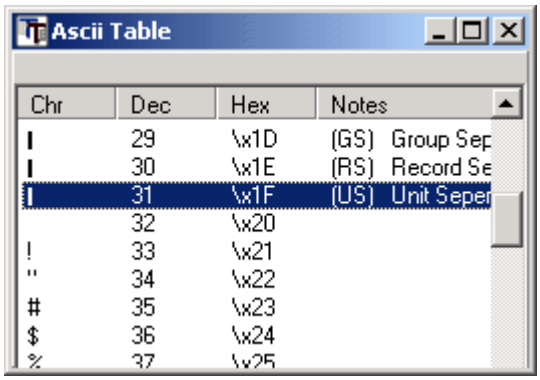
Wenn jetzt auf Ok gedrückt wird und eine Wiederholung von 1 bis unendlich belassen wird, ist das Ergebnis:

```
[-[:word:]€-ÿ!#%&()+./=?@\\~]+
```

Der Bindestrich erscheint am Anfang des Ausdrucks, da er keinen Zeichenbereich definieren soll.

9.2.9.5 ANSI Tabelle

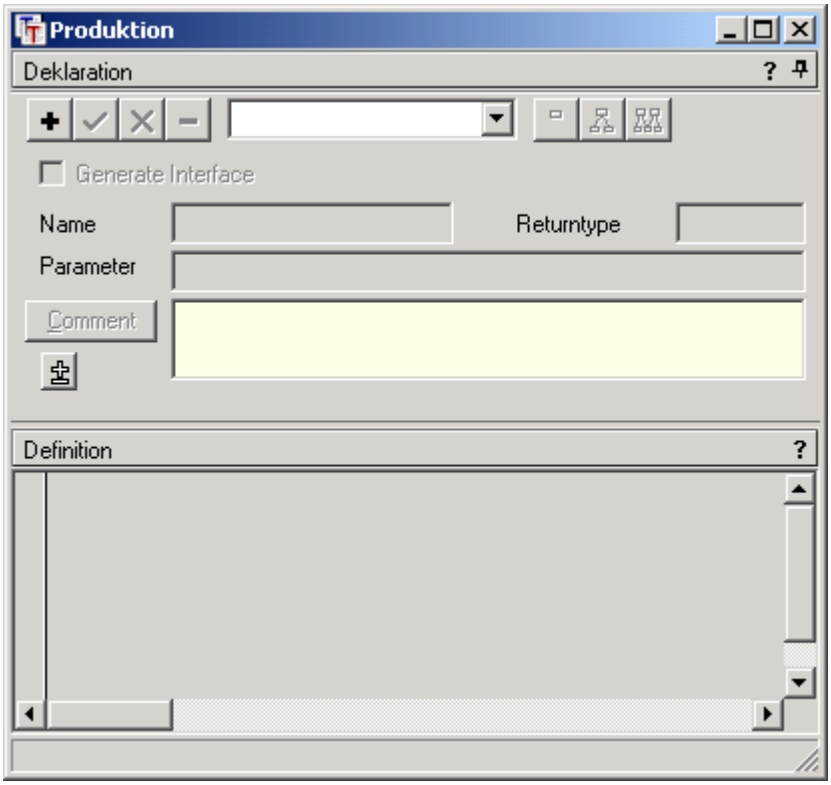
Unter dem Menüeintrag *Hilfe* kann man sich im Unterpunkt *ASCII-Tabelle* eine Liste aller ASCII-Zeichen anzeigen lassen.



In der ersten Spalte ist - soweit darstellbar - das Zeichen selbst zu sehen. In der zweiten Spalte steht der zugehörige Code in dezimaler Schreibweise und in der dritten Spalte in hexadezimaler Schreibweise. In der letzten Spalte gibt es für einige Zeichen eine zusätzliche Anmerkung.

9.3 Verwalten und Parsen

Es gibt vier einander ähnliche Fenster für die vier Typen von Skripten, in denen jeweils ein Skript angezeigt und bearbeitet werden kann. Die Abbildung zeigt als Beispiel das Fenster für eine Produktion:

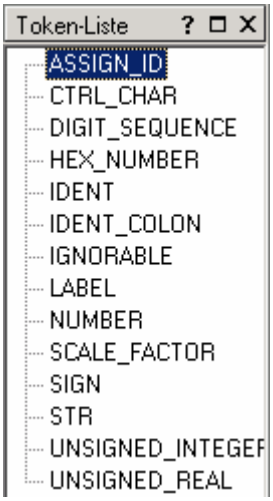


Es gibt jeweils ein solches Fenster

1. Token
2. Produktionen
3. Klassen-Elemente (Variablen und Funktionen)
4. Tests

Die vier Fenster verfügen über eine einheitliche Werzeugleiste und ein entsprechendes Menü, über die Token, Produktionen, Funktionen, Variablen oder Tests verändert, umbenannt, gelöscht oder hinzugefügt werden.

Für die Listen aller in einem Projekt vorhandenen Skripte eines Typs gibt es jeweils ein weiteres Fenster. Wenn einer der Namen in der Liste angeklickt wird, so wird das zugehörige Skript angezeigt. Die Abbildung zeigt als Beispiel das Fenster für die Liste der Token:



Ein Sonderfall ist die **Liste der Produktionen**. Sie wird im Syntaxbaum-Fenster angezeigt. Für kompilierte Produktionen wird hier nicht nur der Namen der Produktion sondern seine gesamte syntaktische Struktur dargestellt.

Da in einem neu angelegten Projekt zunächst keine Regeln vorhanden sind, sind auch die meisten Knöpfe zur Regelverwaltung und die entsprechenden Menüpunkte deaktiviert. Eine erste Regel kann eingefügt werden, indem der Einfügeschalter angeklickt oder der Menüpunkt Projekt | Neu betätigt wird.

9.3.1 Werkzeugleiste und Menü

Die Skriptverwaltungen verfügen alle über ein gleich aussehende Werkzeugleiste.


Neues Skript


Änderungen übernehmen


Änderungen verwerfen


Skript löschen


einzelnes Skript parsen/testen

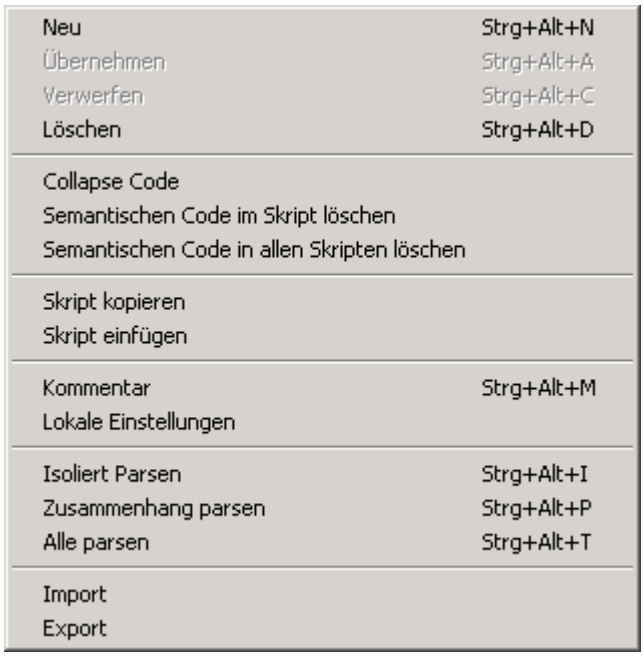

alle Skriptabhängigkeiten parsen/alle Skripte gleicher Gruppe testen


alle Skripte parsen/testen


zum vorherigen Skript

 erneut zu Skript

Die Aktionen, die über die einzelnen Schalter dieser Leiste ausgeführt werden, sind auch als Untermenüs des Menüs "Projekt" ausführbar. Das Projektmenü ist nur dann in das gesamte Hauptmenü eingeblendet, wenn eines der Verwaltungsfenster aktiv ist.



Zusätzlich zu den Aktionen, die durch die Schalter ausgelöst werden gibt es hier Menüpunkte für die lokalen Optionen, zum Import und Export der Skripte und zum Löschen des semantischen Codes. Auf der Interpreter- und der Test-Seite ist der letzte Menüpunkt ersetzt durch die Möglichkeit alle Skripte der jeweiligen Seite zu löschen.

9.3.2 Einfügen

 Neues Skript

Eine neues Skript kann eingefügt werden, indem zunächst der +-Schalter betätigt wird. Nun werden die zuvor grauen Felder weiß dargestellt und sind schreibbar. Ein Skript kann erst dann in das Projekt übernommen werden, wenn ein Name für die Regel vergeben worden ist. Ein Name kann aus alphanumerischen Zeichen und Unterstrichen bestehen, wobei letzterer nicht am Anfang des Namens stehen darf.

Beispiele.: Identifier, Const_declaration, UB40

Jeder neue Name muss sich in mindestens einem Zeichen von allen anderen Namen der in der

Verwaltung enthaltenen Skripte unterscheiden.

9.3.3 Löschen

 Skript löschen

Eine Skript kann gelöscht, d.h. aus der Verwaltung entfernt werden. Hierzu wird sie zunächst in der Liste als aktuelles Skript markiert. Nun wird der Löschen-Schalter [-] betätigt.

Ein Löschvorgang kann nicht rückgängig gemacht werden. Nach dem Abspeichern eines Projekts sind alle zuvor gelöschten Skripte verloren. Wurde das Projekt nach dem Löschen eines Skripts noch nicht abgespeichert, so kann es neu geladen werden, um das gelöschte Skript wiederzuerhalten. In diesem Fall gehen jedoch alle anderen Änderungen verloren, die vor dem erneuten Laden des Projekts vorgenommen wurden. Im Notfall kann TETRA ein zweites mal mit dem Projekt in der alten Form gestartet werden und das betreffende Skript von dort kopiert werden.

9.3.4 Bearbeiten

Ein vorhandenes Script kann unmittelbar ediert werden. Der neue Text kann direkt in das entsprechende Feld geschrieben werden. Dadurch wechselt der texttransformer automatisch in den Editiermodus.

Ob sich ein Skript im Editiermodus befindet, ist an dem Hintergrund des Textfeldes zu sehen.

Texthintergrund:	weiß	=	Editiermodus
	grau	=	kein Editiermodus

Sobald eines der Felder eines Skripts verändert werden, ändert sich auch der Aktivierungsstatus der Schalter in der Werkzeugleiste: der Einfüge- und der Lösch-Schalter werden deaktiviert, während der Bestätigen-Knopf aktiviert wird.

9.3.5 Verwerfen

 Änderungen verwerfen

Ein bearbeiteter Text kann in seinen vorherigen Zustand zurückgesetzt werden solange die Änderungen nicht übernommen wurden. Hierzu dient der Abbruchschalter oder der Menüpunkt "Projekt|Verwerfen".

9.3.6 Übernehmen

 Änderungen übernehmen

Eine neues Skript oder Änderungen an bestehenden Einträgen können durch die Betätigung des

Bestätigenschalters oder des Menüpunktes "Projekt|Bestätigen" in das Projekt übernommen wurden.

Die Hintergrundfarbe eines Skripteditors ist weiß, wenn er sich im Editiermodus befindet. Sobald die Änderungen übernommen sind, ist wechselt die Hintergrundfarbe nach grau.

9.3.7 Umbenennen

Eine Skript kann umbenannt werden. Hierzu wird sie zunächst in der Liste als aktuelles Skript markiert. Nun kann der Text im Namensfeld des Skripts geändert werden. Nach Betätigung des Bestätigen-Schalters

 Änderungen übernehmen

wird das Skript unter dem geänderten Namen in die Verwaltung übernommen, wenn der Name nicht mit einem anderen bereits vorhandenen Namen kollidiert. d.h. wenn der neue Name von allen bisherigen Namen verschieden ist. Andernfalls erscheint ein entsprechender Warnhinweis.

9.3.8 Navigation

Zur Auswahl eines bestimmten Skripts muss dessen Name in der Gesamtliste aller Skripte angeklickt werden.

Doppelklick auf einen der (fett gedruckten) Namen von Regeln, die innerhalb einer Produktion verwendet werden führt zu deren Auswahl.

Mittels der beiden Schalter der Werkzeugleiste

 zum vorherigen Skript

 erneut zu Skript

kann zu dem Skript zurück navigiert werden, das vor dem aktuellen aktiv war, bzw. wieder zum ersten.

9.3.9 einzelnes Skript parsen/testen

 einzelnes Skript parsen/testen

Für Token- und Produktionen wird durch diese Funktion die Syntax des aktuellen Skripts geparst. Hierdurch können Syntaxfehler im Text des Skripts aufgespürt werden.

Für Klassen-Elemente ist diese Funktion inaktiv.

Ein einzelner Test wird für ein Testskript ausgeführt

9.3.10 Zusammenhang (als Startregel) parsen/testen

 alle Skripte mit parsen, von denen das aktuelle abhängt

Für Token hat dieser Schalter keine besondere Bedeutung; da die Token nicht voneinander abhängen. Bei Betätigung wird wie beim vorherigen Schalter ein einzelner regulärer Ausdruck geparst.

Bei Produktionen werden hierdurch neben dem aktuellen Skript alle Skripte geparst, von denen das aktuelle abhängt. So wird eine tiefer gehende Prüfung der formalen Richtigkeit dieser Skripte durchgeführt. Getestet werden:

Syntaktische Richtigkeit der Skripte
Vollständigkeit der Skripte
Syntaktische Richtigkeit des Interpretercodes
Typüberprüfung des Interpretercodes

Für Klassen-Elemente ist diese Funktion inaktiv.

Im Test-Fenster werden alle Testskripte ausgeführt, die zur gleichen Gruppe gehören wie das aktuelle Skript.

9.3.11 Alle Skripte parsen/testen

 alle Skripte parsen

Auf der **Token- und der Produktionenseite** werden durch diese Funktion alle Skripte geparst, so dass sie sämtlich auf syntaktische Richtigkeit und Vollständigkeit geprüft werden.

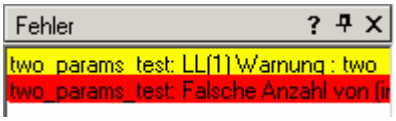
Für die Produktionen hat dies den zusätzlichen Effekt, als sei die Option zum Erzeugen eines Interfaces für alle Produktionen gesetzt. Automatisch wird für jede Produktion ein Scanner erzeugt, der das erste Token eines gewählten Textabschnitts testen kann, so dass zwischen sämtlichen Produktionen ein interaktiver Wechsel möglich ist.

Auf der **Elementseite** werden hiermit alle Funktionsskripte Variablendeklarationen und -initialisierungen auf syntaktische Richtigkeit überprüft.

Auf der **Testseite** werden mit dieser Funktion alle Tests ausgeführt. Hierbei werden jeweils die Tests einer Gruppe gemeinsam geparst.

9.3.12 Fehler-Meldungen

Fehlermeldungen und Warnhinweise die beim Parsen eines Skripts auftreten werden im *Fehler* -Fenster angezeigt.



Die Einträge in dem Fenster sind farblich markiert:

gelbe Einträge sind Warnhinweise
rote Einträge sind Fehlermeldungen

Nach einem **einfachen Klick** mit der linken Maus auf einen der Einträge wird die vollständige Meldung in dem gelblich hinterlegten Meldungsfenster in der Mitte der Entwicklungsumgebung angezeigt. Über **F1** kann die **Hilfe** zu der Art des Fehlers aufgerufen werden.

Nach **Doppelklick** auf eine der Zeilen wird die Position im Skript angezeigt, die die Meldung veranlasst hat.

9.3.13 Semantischen Code löschen

Mit der Funktion: semantischen Code löschen, können alle semantischen Aktionen aus einem Skript entfernt werden. Darin eingeschlossen sind Parameter und Rückgabewerte.

Auf der Token-Seite werden in allen Skripten die Parameter, Rückgabewerte und Aktionen gelöscht.
Auf der Produktionen-Seite kann die Funktion entweder für ein einzelnes Skript aufgerufen werden oder für sämtliche Skripte der Seite.
Auf der Element-Seite gibt es nichts anderes als semantischen Code. Deshalb besteht hier die Möglichkeit alle Skripte aus dem Projekt zu entfernen.
Auf der Test-Seite besteht ebenfalls die Möglichkeit alle Skripte aus dem Projekt zu entfernen.

Vom allgemeinen Bearbeiten Menü aus kann der semantische Code der Token- der Produktionen- und der Interpreter-Seite auf einmal entfernt werden.

Beispiel:

Wird die Funktion auf das folgende Skript angewandt:

```
Parameter: str& xs
Rückgabetyp:
Text:
  {{
```

```
  m_bInNewLine = true;
  node n("ClearItpText");
  }}
  Expression[n]
  {{
  xs = m_ftClear.visit(n);
  }}
  {-
  out << xs;
  -}
```

so erhält man:

Parameter:
Rückgabetyp:
Text:
```
  Expression
```

9.3.14 Import

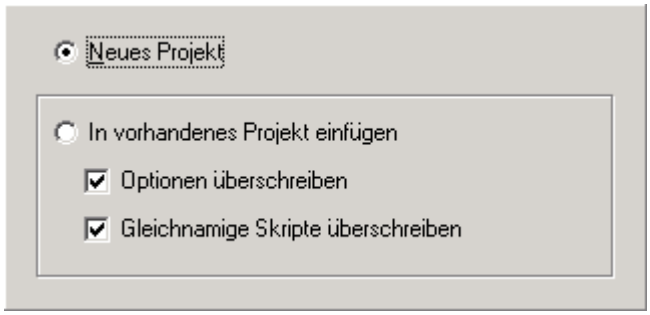
Über die Importfunktion können Skriptlisten importiert werden, die zuvor aus TETRA exportiert wurden.

Je nach Art des Skripts haben diese Dateien ein unterschiedliches Format und sind durch unterschiedliche Dateierweiterungen gekennzeichnet:

Skripttyp	Erweiterung
Alle	tti
Token	ttx
Produktionen	ttr
Interpreter	tte
Test	ttt

Es **wird nicht überprüft, ob** die importierten Token, Produktionen usw. der **korrekten TETRA-Syntax** gehorchen. Im Gegenteil: die Syntax für den Import ist möglichst tolerant gestaltet. Die meisten Syntaxelemente sind optional. So wird der Import von Grammatiken erleichtert, die mit anderen Parsergeneratoren verfasst wurden. Exakte Konvertierungsprogramme für derartige Grammatiken zu schreiben ist ein unverhältnismäßig großer Aufwand. Die Möglichkeiten der IDE vereinfachen die Anpassung an die TETRA-Syntax enorm.

Falls bereits ein Projekt im TextTransformer geöffnet ist besteht beim Import die Auswahl, ein neues Projekt zu erstellen oder die importierten Skripte in das bereits bestehende Projekt mit einzufügen.



Das Kästchen "**Optionen überschreiben**" bestimmt, ob die Projektoptionen aus der Import-Datei übernommen werden, oder nicht. Wenn das Häkchen gesetzt ist, werden die aktuellen Projektoptionen auch dann überschrieben, wenn in der Import-Datei keine Optionen angegeben sind. Es werden dann die Default-Optionen gesetzt.

Wenn die Option "**Gleichnamige Skripte überschreiben**" **nicht** aktiviert ist, so werden alle diejenigen Skripte der Verwaltung zugefügt, deren Namen von den Namen der bereits vorhandenen Skripte unterschieden ist. Skripte gleichen Namens werden beim Import dann nicht überschrieben.

Das Import-Format für ein gesamtes Projekt hat die Struktur:

```
ImExport ::=
             "TextTransformer"
             ProjectOptions

             (
               Tokens
             | Productions
             | Members
             | Tests
             )*
```

Die Formate für den Export der Token, Produktionen, Interpreter-Skripte und der Tests ist identisch mit dem der jeweiligen Produktionen des Gesamt-Formats.

```
Tokens ::= "TOKENS" Token*
Productions ::= "PRODUCTIONS" Production*
Members ::= "MEMBERS" Member*
Tests ::= "TESTS" Test*
```

Die Details können dem beigefügten ImExport-Projekt entnommen werden. Als exemplarisches Beispiel sei hier noch die Struktur einer Produktion angeührt:

```
Production ::=
               Comment?
               IDENT
               Params?
               ReturnType?
               "::="
               LocalOptions?
               Field
```

Beispiel:

```
/*
*/
Text(  ) : void
[LocalOptions]
CaseSensitive=1
CommentToCode=0
CreateInterface=0
Exportable=1
GlobalLiteralScanner=1
GlobalRegexScanner=0
IgnoreChars=IGNORE
IgnoreWhiteSpace=1
InclusionProd=
Interpretable=1
IsNullableWarning=1
Separated=1
StartSuccNullableWarning=1
TestAllLiterals=0
UseIgnoreRegex=1
UseLocalOptions=1

(>
"(>"
(
    SKIP
  | STRING
)*
"<)"
<)
```

Die Definition einer Produktion beginnt mit ihrem Namen, auf den dann optional in Klammern gesetzte Parameter folgen und wiederum optional ein Rückgabewert hinter einem Doppelpunkt. Der Text der Produktion wird in die Klammern "(>" und "<)" eingeschlossen. (Diese Zeichen sind so gewählt, dass sie innerhalb des Textes nicht auftauchen dürften.) Die Syntax für die anderen Skripte ist analog, wobei die Anzahl der folgenden Texte gleich der Anzahl der jeweiligen Eingabefelder im TextTransformer ist.

Auch die Angabe der **Optionen** ist optional. Für nicht genannte Optionen wird der jeweilige Default-Wert gesetzt.

9.3.15 Export

Über die Exportfunktion werden die Skripte in eine Ascii-Datei geschrieben.

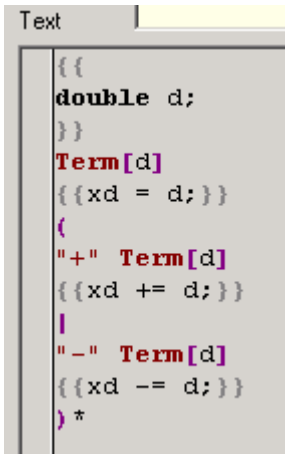
Die exportierten Dateien können der Sicherung oder dem Reimport in andere Projekte dienen.

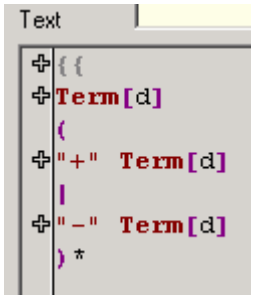
9.3.16 Semantischen Code einklappen

Auf der Produktionen-Seite gibt es den Schalter:

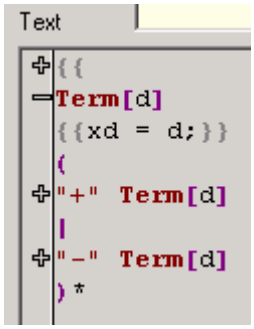
 Semantischen Code einklappen

Dies dient der übersichtlichen Darstellung des syntaktischen Teils der Regel.
Der semantische Code ist im Text zunächst dadurch gekennzeichnet, dass er in jeweils einem der Klammerpaare {{...}}, {_..._}, {-...-} oder {=...=} eingeschlossen ist. In folgendem Beispiel sind dies jeweils {{...}} Klammerungen (linkes Bild).
Wird Schalter zum Einklappen des semantischen Codes erstmals betätigt, so werden sämtliche der genannten Klammerausdrücke, durch ein Pluszeichen am linken Rand des Editierfeldes ersetzt (rechtes Bild).





Die Pluszeichen befinden sich jeweils in der Zeile über dem zugehörigen Klammerausdruck. Das ist bei der ersten semantische Aktion des Beispiels nicht möglich, da sie in der obersten Zeile beginnt. Die Einklappung ist daher unvollständig. Auch, wenn sich semantischer Code mit dem anderen Code eine Zeile teilt, ist die Einklappung nicht möglich.
Durch Mausklick auf eines der Pluszeichen kann nun die zugehörige semantische Aktion einzeln wieder hervorgeholt werden:



9.4 Debuggen und Ausführen

Die Regeln und regulären Ausdrücke, die in der Verwaltung definiert sind können im TETRA-Programm direkt getestet und ausgeführt werden. Zu diesem Zweck ist zunächst ein Quelltext zu laden, der gemäß den Regeln analysiert werden soll. Nun kann in der Auswahlbox der Werkzeugleiste eine Startregel ausgewählt werden.

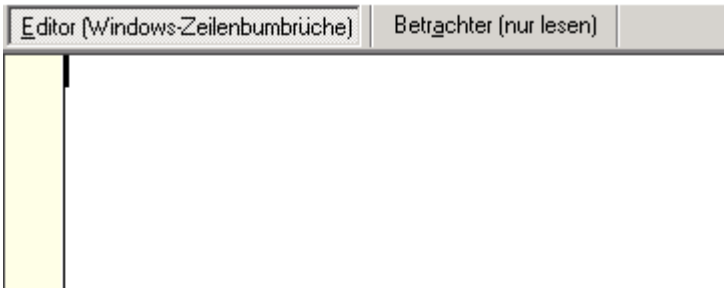
Das Programm (die Startregel) kann in verschiedenen Modi ausgeführt werden: schrittweise oder in einem Zug. In jedem Fall wird zunächst die Startregel geparst und samt ihren Unterregeln in einer Baumstruktur dargestellt.

Es gibt zahlreiche Hilfsmittel, die das **Debuggen** (Fehler-Suchen) von TextTransformer-Projekten erleichtern:

- Der erkannte und der erwartete Textabschnitt werden im Text markiert
- Haltepunkte können im Text gesetzt werden
- Haltepunkte können ebenfalls im Syntaxbaum gesetzt werden
- Der Inhalt von Variablen kann im Variablen-Inspektor betrachtet werden
- Es kann zurück zur aktuellen Position gesprungen werden
- Die zuletzt abgerufene Information wird in der Info-Box gezeigt
- Debug-Informationen werden im Log-Fenster ausgegeben
- Erkanntes und erwartetes Token werden angezeigt
- Ein Stack der aufgerufenen Produktionen wird angezeigt
- Die Anfängermengen von Produktionen und Verzweigungen werden angezeigt

9.4.1 Quelltext

Quelltextfenster:



Am oberen Rand des Quelltextfensters kann zwischen einem reinen Text-Betrachter und einem Editor gewählt werden. Der jeweils dargestellte Text ist der Quelltext, der analysiert und transformiert wird.

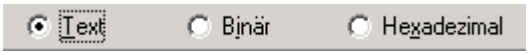
Editor (Windows-Zeilenumbrüche)

Nach dem Start des TextTransformers ist zunächst der Editor aktiv. So ist es möglich direkt einen Text in das Fenster einzugeben oder einen Text hineinzukopieren. Man kann den Parser mit diesem Text und beliebigen Variationen des Textes testen

Betrachter (nur lesen)

Normalerweise wird man einen auf der Festplatte vorhandenen Quelltext parsen wollen. Wird eine solche Datei geöffnet, so erscheint sie stets im Betrachter. Der Betrachter zeigt den Text genau so, wie er auf der Festplatte gespeichert ist. Auch Textdateien mit nicht Windows-konformen Zeilenumbrüchen und sogar Binärdateien werden korrekt angezeigt und können geparst werden. Jedoch ist es nicht möglich, den Text zu verändern.

Der Betrachter kann die geladene Datei in drei Modi anzeigen:



Text

Im Textmodus werden die Zeilenumbrüche wie gewohnt dargestellt.

```
%PDF-1.2
%âãÏÓ
4480 0 obj
<<
/Linearized 1
```

Binär

Im Binärmodus werden die Zeilenumbrüche und ander Kontrollzeichen durch Punkte dargestellt.

```
%PDF-1.2.%âãÏÓ..4480 0 obj.<< ./Linearized 1 ./O
2997 ./E 103570 ./N 394 ./T 1473277 .>> .endobj.
```

Hexadezimal

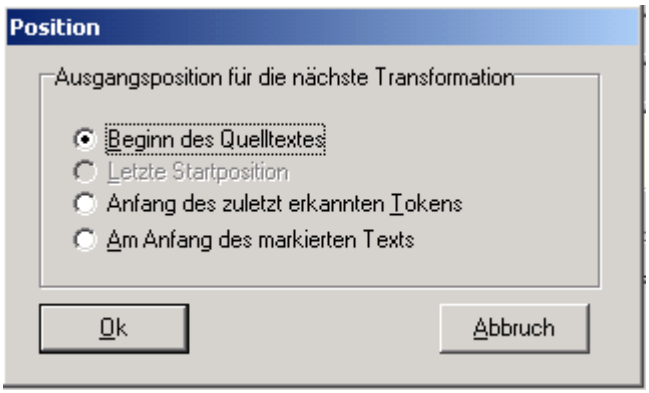
Im Hexadezimalmodus werden neben den Zeichen auch ihre hexadezimalen Werte angezeigt.

```
25 50 44 46 2D 31 2E 32 | 0D 25 E2 E3 CF D3 0D 0A | %PDF-1.2.%âãÏÓ..
34 34 38 30 20 30 20 6F | 62 6A 0D 3C 3C 20 0D 2F | 4480 0 obj.<< ./
4C 69 6E 65 61 72 69 7A | 65 64 20 31 20 0D 2F 4F | Linearized 1 ./O
```

9.4.2 Textabschnitt wählen

Der Textteil des Quelltextfensters, der von TETRA bearbeitet wird hängt vom aktuellen Zustand des Programms und von einer Wahl des Benutzer ab. In den Einstellungen der Benutzeroberfläche kann gewählt werden, ob eine Transformation prinzipiell am Beginn des Eingabetextes anfangen soll, oder ob auch andere Möglichkeiten zugelassen werden sollen. Im letzteren Fall erscheint vor dem Start

des Debuggers der folgende Auswahldialog:



Im Editor wird die Option **Am Anfang des markierten Textes** auch angeboten, wenn kein Text markiert ist. In diesem Fall wird bei der aktuellen Position des Cursors gestartet.

9.4.3 Aktionen aktivieren

Zum isolierten Testen allein der Analyse eines Eingabetextes kann der Interpreter abgeschaltet werden. Soll hingegen der Interpreter-Code ausgeführt werden, d.h. soll der Text transformiert werden, so müssen die Aktionen aktiviert sein.

Das Ein- und Ausschalten des Interpreters, d.h. die Aktivierung und Deaktivierung der Aktionen kann entweder über das Menü (*Start->Aktionen*) oder über die Checkbox in der Werkzeugleiste erfolgen:

Aktionen eingeschaltet

Aktionen ausgeschaltet

Verwendet der Parser semantische Aktionen in IF- oder WHILE-Strukturen, müssen die Aktionen aktiviert bleiben, da das Projekt sonst nicht kompiliert..

9.4.4 Startregel wählen

Eine Texttransformation ist nur möglich, wenn eine der Regeln des Projekts als Startregel ausgewählt wurde. Diese Auswahl erfolgt über die Combobox in der Werkzeugleiste.

Standardmäßig steht nach dem Laden eines Projekts die in den Projektoptionen angegebene Regel im Auswahlfeld der Box. Falls in den Projektoptionen keine Startregel explizit gesetzt ist wird diejenige Regel als Startregel angezeigt, deren Namen mit dem des Projekts übereinstimmt. Gibt es

eine solche Regel nicht, bleibt die Auswahlbox leer und es muss manuell eine Startregel gesetzt werden.

Auch Produktionen, die Parameter benötigen, können als Startregeln gewählt werden. Bei der Ausführung der Regel werden dann Default-Werte für die Parameter eingesetzt.

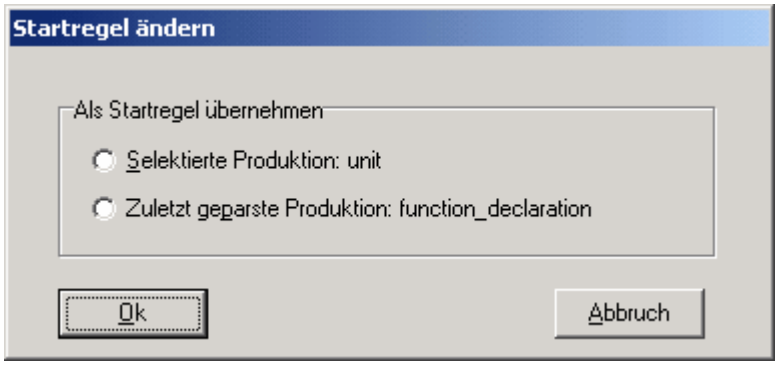
9.4.5 Interaktiver Wechsel der Startregel

Sollen z.B. verschiedene Textabschnitte **interaktiv** mit verschiedenen Produktionen transformiert werden, so kann über die Auswahlbox in der Werkzeugleiste die jeweilige Produktion für den neuen Abschnitt gewählt werden.
Wurden die Produktionen mit Alle Skripte parsen kompiliert, so ist ein interaktiver Wechsel unmittelbar möglich. Wurden sie hingegen mit Zusammenhang parsen kompiliert, was auch der Fall ist, wenn auf der Arbeitsseite unmittelbar mit der Transformation begonnen wurde, so muss die neu gewählte Produktion eventuell zunächst noch (automatisch) kompiliert werden.
Was genau bei einem Wechsel der Startregel passiert, hängt davon ab, was zuvor getan wurde.

9.4.6 Wechsel der Startregel

Was genau bei einem Wechsel der Startregel passiert, hängt davon ab, was zuvor getan wurde. In einer längeren Arbeitssitzung am Texttransformer können mehrere Situationen auftreten:

1. **Seitenwechsel**: Auf der Produktionen-Seite wurde zuletzt eine andere Produktion geparst (kompiliert) als die in der Auswahlbox selektierte Startregel. Wird nun zur Arbeits-Seite gewechselt und wird einer der Schalter zum Ausführen des Programms betätigt, so muss zunächst in folgender Box eine Auswahl getroffen werden:



2. **Wechsel der Startregel über die Auswahlbox**: Wurden die Produktionen zuvor mit Alle Skripte parsen kompiliert, so ist ein Wechsel ohne weiteres möglich. Wurden hingegen eine Startregel mit Zusammenhang parsen kompiliert, erscheint obige Box, um den Wechsel zu bestätigen.

Sowohl unter Punkt eins als auch unter Punkt zwei gibt es zwei Möglichkeiten(, wenn mit Zusammenhang parsen kompiliert wurde):

a) **Neue Regel ist bereits kompiliert:** Die neu gewählte Regel gehört zum Ensemble derjenigen Produktionen, von denen die erste Startregel abhängt. Dann ist die neue Regel ebenfalls bereits kompiliert. Eine unmittelbare Ausführung dieser neuen Regel ist aber nur dann möglich, wenn für sie die Interface-Option aktiviert ist. Ist diese Option nicht gesetzt, so muss die neue Startregel zunächst nochmals kompiliert werden. Hierbei wird ein gesonderter Scanner erzeugt, der die Menge der Token umfasst, mit denen die Regel beginnen kann und der den Anfang des zu parsenden Textes auf diese Token hin testet.

b) **Neue Regel ist noch nicht kompiliert:** Die neu gewählte Regel gehört nicht zum Ensemble derjenigen Produktionen, von denen die erste Startregel abhängt. In diesem Fall ist eine Neukompilierung unabdingbar.

9.4.7 Startregel parsen

Die in der Auswahlbox der Werkzeugleiste gewählte Produktion kann direkt geparst werden, indem der Schalter (der gleichen Werkzeugleiste)

betätigt wird. Befindet man sich nicht auf der Tetra-Seite, so erfolgt ein Wechsel auf die Produktionen-Seite.

9.4.8 Syntaxbaum

Wenn eine Startregel geparst wurde, wird sie samt den ihren Unterregeln in einer Baumstruktur dargestellt. Über ein Popup-Menü können Haltepunkte auf Baumknoten gesetzt werden oder man kann sich zugehörige Anfängermengen anzeigen lassen.

Zunächst erscheinen die Produktionen in kontrahierter Form:

kontrahierte Darstellungsform

In kontrahierter Form sind lediglich die Namen der Regeln mit einem vorangestellten '+' zu sehen. (Wenn noch keine der Regeln geparst wurde, fehlt das '+'.)

expandierte Darstellungsform

In expandierter Form ist dem Namen ein '-' vorangestellt und unter dem Namen wird die Aufeinanderfolge der in der Regel vorkommenden Unterregeln und Token angezeigt, sowie spezielle Unterknoten für Optionen, Wiederholungen und Alternativen.

Knoten-Icon

Jedem Knoten ist ein kleines Icon vorangestellt, dass den Typ des Knotens kennzeichnet. Den Namen der Knoten wird jeweils ein Prefix vorangestellt, der ebenfalls den Typ bezeichnet.

Icon	Typ	Prefix
	Produktion	
	mit lokalen Optionen	
	Aufruf eine Produktion	NT
	Terminalsymbol = Token	T
	ANY-Symbol	ANY
	SKIP-Symbol	SKIP
	Alternative	Alt
	Option	Opt
	Wiederholung	Rep
	optionale Wiederholung	OptRep
	gezählte Wiederholung	Count
IF	IF-Struktur	If
W	WHILE-Struktur	Cond
	BREAK	Br
	Semantische Aktion	Sem

Verkettung

Regelbestandteile die innerhalb der Regel aufeinander folgen werden im Baumdiagramm als untereinander stehende Knoten dargestellt, die durch eine vertikale Linie mit kleinen Pfeilsymbolen miteinander verbunden sind. Eine Produktion

A = a b c d

wird dann folgendermaßen angezeigt

Alternativen

Für Regelbestandteile, die alternativ zueinander sind, wird ein Knoten eingefügt, der die Alternativen zusammengefasst und unterhalb dieses Knotens werden die einzelnen Alternativen als diskrete Knoten, also unverbunden dargestellt. Eine Produktion

A = a | b | c | d

wird dann folgendermaßen dargestellt

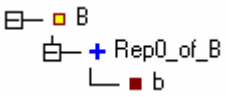
Der Name des die Alternativen zusammenfassenden Knotens wird aus dem Prefix "Alt" gebildet, dem ein Zähler folgt, der die Alternativen der Regel beginnend mit Null hochzählt. Hieran angehängt wird "_of_" und der Namen des übergeordneten Knotens. Aus dem Namen lässt sich somit der Typ und die Position des Knotens innerhalb der gesamten Regelsystems rekonstruieren.

Optionen und Wiederholungen

Für Optionen oder Wiederholungen wird analog ein übergeordneter Knoten gebildet, dessen Kind-Knoten die eigentliche Option oder Wiederholung ist. Die Produktion

B = (b)+

sieht dann folgendermaßen aus:



Der Name des Options- oder Wiederholungs-Knotens wird analog zu der Namensgebung bei Alternativen gebildet, wobei die Ausdrücke

Opt	für Optionen
0ptRep	für null und mehrmalige Wiederholungen
Rep	für ein- und mehrmalige Wiederholungen

verwendet werden.

semantische Aktionen

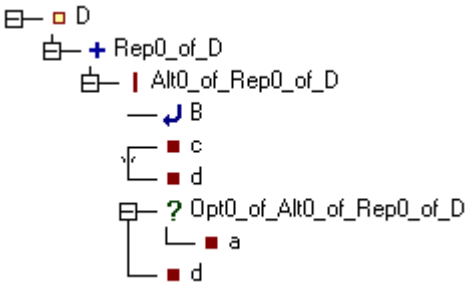
Für Positionen an denen im Skript einer Produktion semantische Aktionen ausgeführt werden, werden im Baumdiagramm einfache Knoten angezeigt, deren Namen wiederum aus dem Namen der Produktion gebildet werden, indem der Ausdruck "_Sem" und ein Zähler angehängt werden. Die den Token direkt zugeordneten Aktionen werden nicht gesondert angezeigt.

komplexes Beispiel

Besteht eine der Alternativen selbst aus einer Folge von Token oder Regeln, so werden diese Folgen als untereinander verbundene Knoten dargestellt. Die gesamte Gruppierung dieser Alternative ist von den übrigen Alternativen getrennt. Eine etwas komplexere Produktion könnte z.B. lauten:

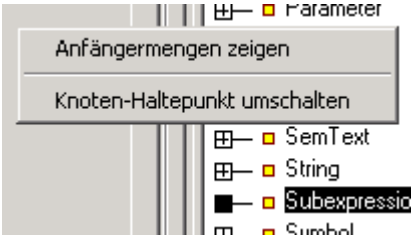
D = (B | c d | (a)? d)+

Im Syntaxbaum sähe sie folgendermaßen aus:



9.4.8.1 Popup-Menü

Zu jedem Knoten des Syntaxbaums lässt sich mit der der rechten Maustaste ein lokales Popup-Menü aufrufen.



Über den ersten Menüpunkt:

Anfängermengen zeigen

erhält man Informationen über den Kontext des Knotens und über den unteren Menüpunkt:

Knoten-Haltepunkt umschalten

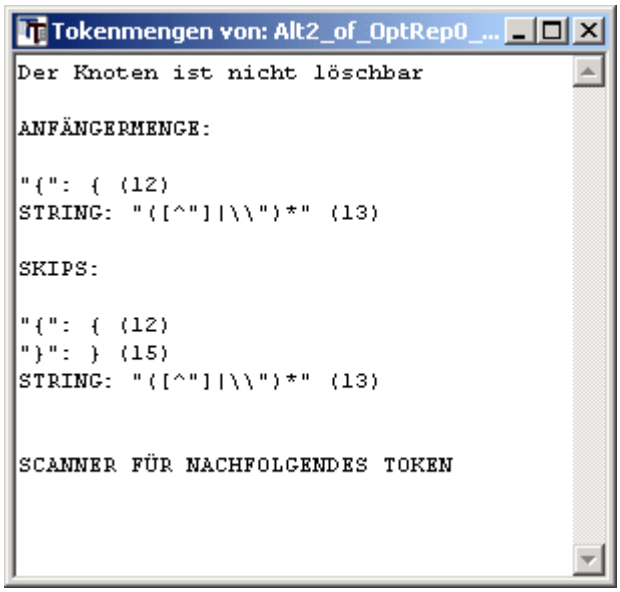
kann bestimmt werden, ob die Ausführung des Debuggers an dem Knoten Halt machen soll oder nicht.

Die Aktivierung dieser Menüpunkte ist nur sinnvoll, wenn der Knoten kompiliert ist.

9.4.8.2 Anfängermenge zeigen

Über das Popup-Menü, das zu einem Knoten erscheint, wenn die rechte Maustaste gedrückt wurde, kann man sich Informationen über die Anfängermengen des Knotens anzeigen lassen, vorausgesetzt, er ist geparst.

Die Anzeige sieht beispielsweise folgendermaßen aus:

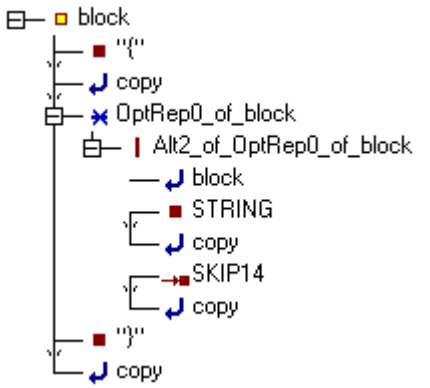


Anmerkung:
Zusätzlich wird ab der Version 0.9.8.8 eine Zeile mit den für den Knoten gültigen Optionen angezeigt, z.B.: `Options: sep !icase, global ig lit (!all) !rgx`. Die Bedeutung dieser Abkürzungen wird unten erklärt.

Diese Box gehört zum Knoten für die Alternative in der *block-Produktion* des guard-Beispiels:

```
block =
        "{" copy_text
        (
          block
        | STRING copy_text
        | SKIP copy_text
        )*
        "}" copy_text
```

In der Baumansicht:



1. Titel

Im Titel der angezeigten Box steht der Name des Knotens.

2. Kopfzeile

In der ersten Zeile des Fensters steht, ob der Knoten löschbar ist oder nicht.

3. ANFÄNGERMENGE:

Mit "Anfängerliste" ist eine Liste überschrieben, deren Zeilen jeweils aus dem Namen, dem Text und der (automatisch vergebenen) Nummer von Token bestehen.

```
"{": { (12)
STRING: "([^"]|\\")*" (13)
```

Diese Liste enthält die eigentliche Anfängermenge des Knotens, d.h. alle Token, mit denen die alternativen Ketten, die in dem Knoten ihren Ursprung haben beginnen.

Im Beispiel handelt es sich um den Knoten, der die folgende Alternative darstellt:

```
  block
| STRING copy_text
| SKIP copy_text
```

"{" ist das einzige Token mit dem die *block-Produktion* beginnen kann und
STRING ist ein alternatives Terminalsymbol.

4. SKIPS:

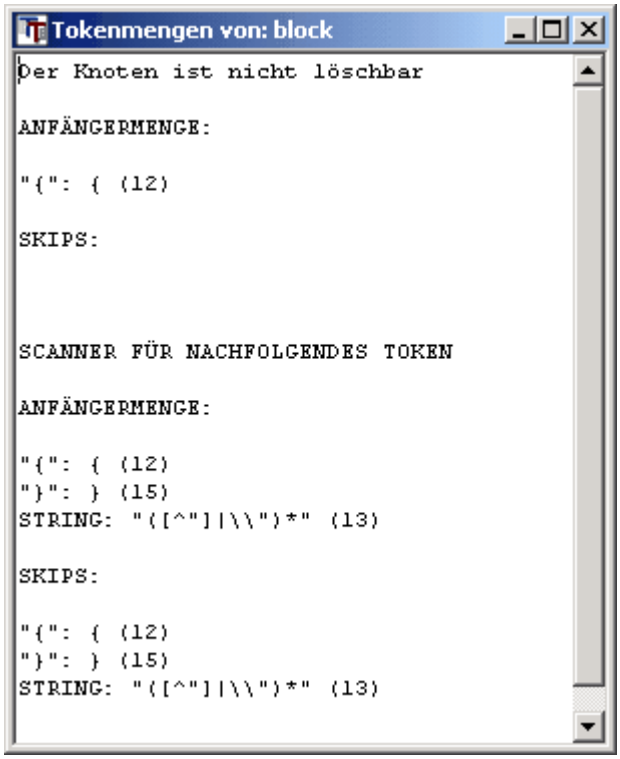
Falls zu den Alternativen des Knotens ein SKIP-Symbol gehört oder falls der Knoten selbst für ein SKIP-Symbol steht, wird dessen Tokenmenge hier angezeigt.
Im Beispiel:

```
"{": { (12)
"}": } (15)
STRING: "([^"]|\\")*" (13)
```

Im Quelltext wird nach der nächsten Position gesucht, an der eines dieser Token vorkommt, wenn an der aktuellen Textposition keines der unter 3. genannten Token gefunden wird.

5. SCANNER FÜR NACHFOLGENDES TOKEN

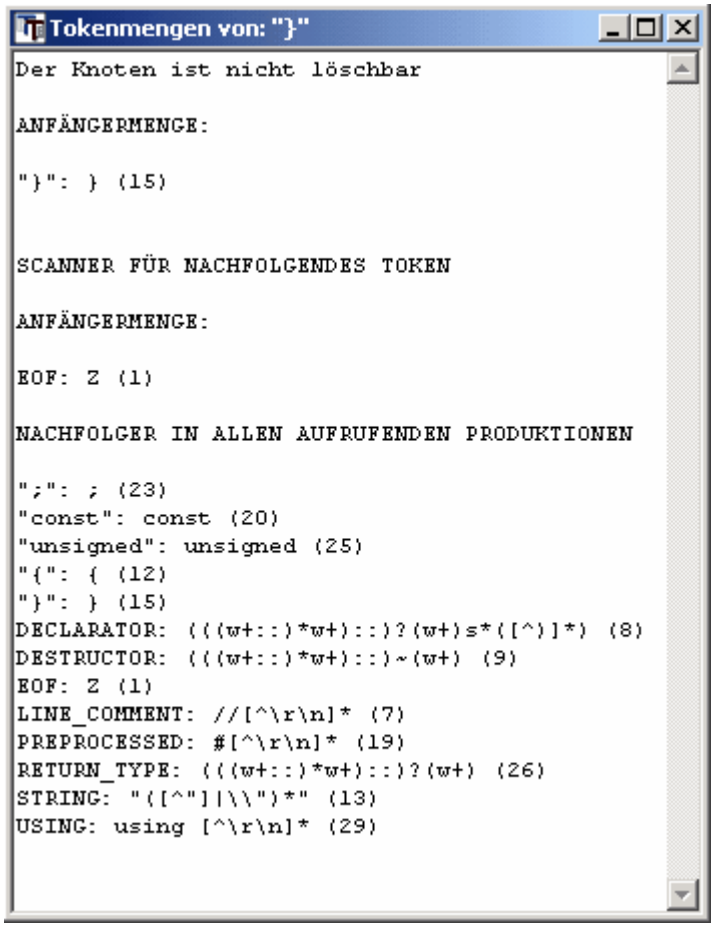
Nur, wenn es sich bei dem Knoten, dessen Informationen angezeigt werden, um ein Terminalsymbol oder um den Aufruf eines Nonterminals handelt, folgen weitere Tokenlisten für den "Scanner für nachfolgendes Token". Während in den Listen unter Punkt 3 und 4 die Token angeführt wurden, deren Erkennung zum Knoten führt, werden in den nachfolgenden Listen die Token angeführt, die zum nächsten Knoten leiten.
In der gleichen *block*-Produktion des obigen Beispiels sieht die Anzeige für den *block*-Knoten folgendermaßen aus:



Nach dem rekursiven Aufruf der *block*-Produktion innerhalb ihrer selbst, muss nach dem nächsten Token im Text gesucht werden. Die Kandidaten der Suche sind in den Listen des Scanners für die nachfolgenden Token genannt. Entweder beginnt mit "{" sofort ein neuer Block oder es folgt ein String oder der aktuelle Block wird mit "}" verlassen. Trifft keiner der drei Fälle an der aktuellen Textposition zu, wird - wegen des SKIP-Symbols - im Text die nächste Stelle gesucht, an der einer der Fälle zutrifft.

6. NACHFOLGER IN ALLEN AUFRUFENDEN PRODUKTIONEN

Schließlich gibt es noch eine weitere Liste, falls der zugehörige Knoten zu einem Terminalsymbol gehört, das "am Ende" einer Produktion steht. "Am Ende" heißt, dass es zu dem Terminalsymbol ein nachfolgendes Token gibt, das sich nicht mehr in der gleichen Produktion befindet. In der *block* -Produktion trifft dies auf den Knoten für die schließende Klammer "}" zu. Die Anfängermengen für diesen Knoten sehen wie folgt aus:



In der Liste der "NACHFOLGER IN ALLEN AUFRUFENDEN PRODUKTIONEN" befinden sich alle Token, die auf die Aufrufe der block-Produktion folgen können. Einen solchen Aufruf gibt es in der *block*-Produktion selbst. Deshalb müssen in der Liste aller token, die auf "}" folgen können auch alle die Token vorkommen, die auf diesen Aufruf folgen können. Diese Token wurden bereits unter Punkt 5 (SCANNER FÜR NACHFOLGENDES TOKEN**)** betrachtet:

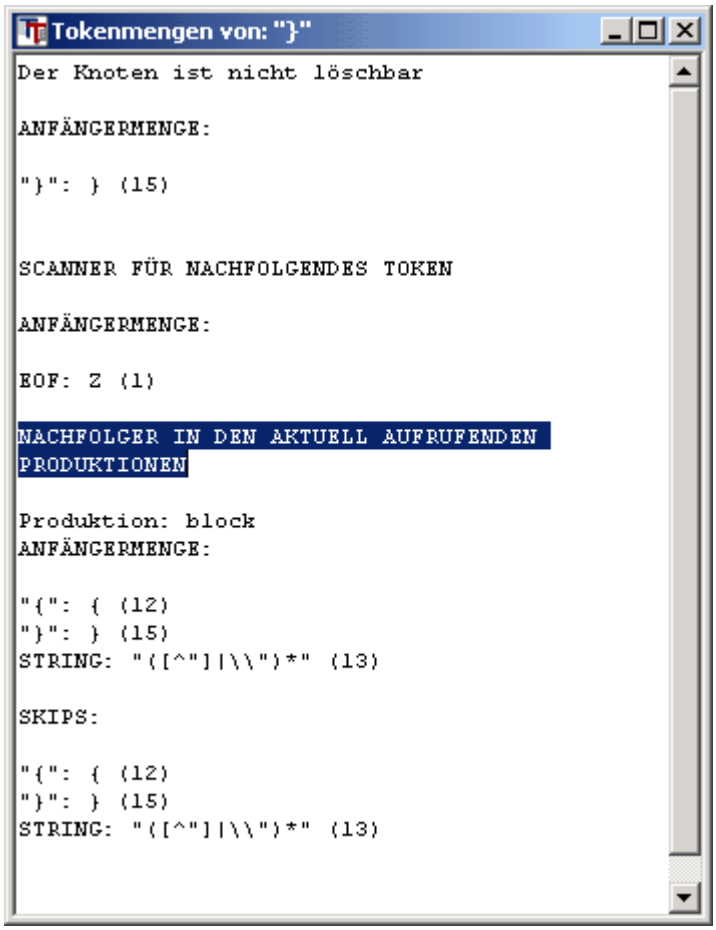
"{": { (12)
"}": } (15)
STRING: "([^"]|\\")*" (13)

Die restlichen "NACHFOLGER IN ALLEN AUFRUFENDEN PRODUKTIONEN**"** folgen auf Aufrufe der *block*-Produktion innerhalb anderer Regeln.

Die letzte Liste sieht anders aus, wenn sie während des Debuggens des *guard*-Projekts angezeigt wird. Sie wird deshalb umbenannt in:

7. NACHFOLGER IN DEN AKTUELL AUFRUFENDEN PRODUKTIONEN

Jetzt werden nur noch diejenigen Token aufgelistet die tatsächlich im Moment folgen können, d.h. die Nachfolger des aktuellen Aufrufs der Produktion. Im Beispiel ist die aufrufende *block*-Produktion hier wiederum die *block*-Produktion selbst.:



Jetzt werden nur noch die unter Punkt 5 besprochenen Token angezeigt. Token die auf andere Aufrufe der *block*-Produktion folgen können sind aktuell irrelevant.

Optionen:

Zusätzlich wird ab der Version 0.9.8.8 eine Zeile mit den für den Knoten gültigen Optionen angezeigt.

Zeichen	Abkürzung für	Bedeutung
!		nicht
sep	separated	Wortgrenzen
icase	ignore case	Gross-/kleinschreinbung nicht beachten
global		globale Scanner-Einstellung
ig	ignore	auszulassende Zeichen
lit	literals	literale Token
all		alle testen
rgx	regular expression	nicht literale Token

Z.B.:

```
Options: sep !icase, global ig lit (!all) !rgx
```

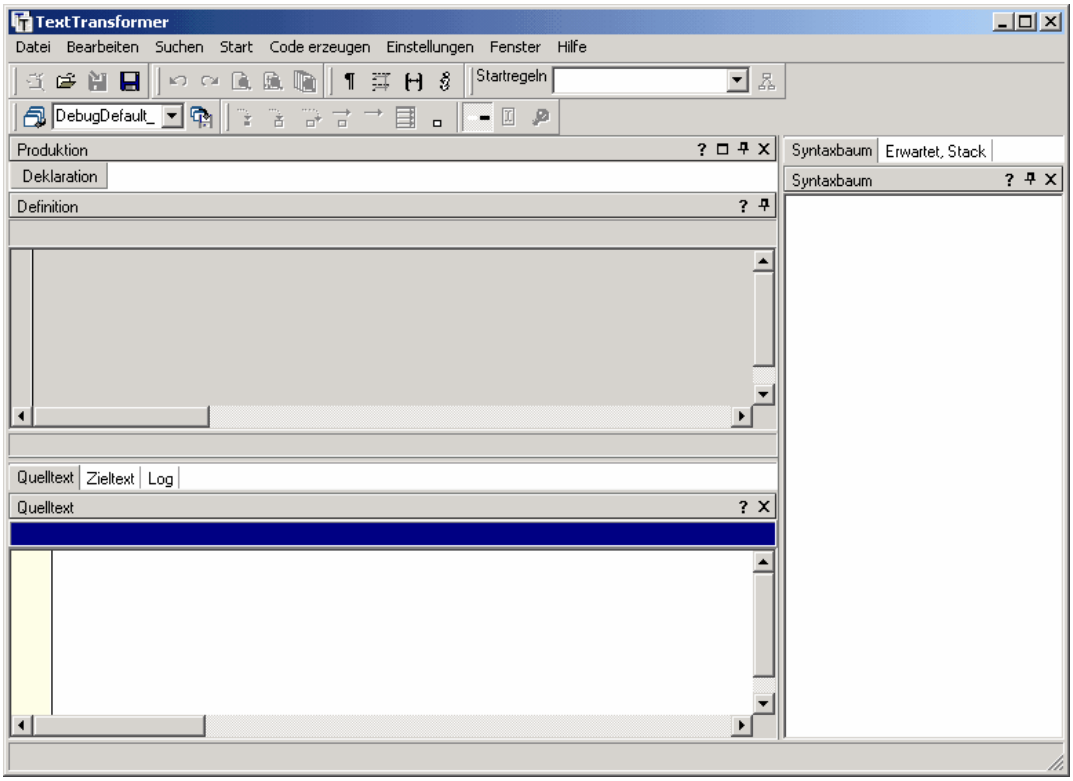
bedeutet, dass literale Token abgetrennte Worte sein müssen, dass Gross-/Kleinschreibung beachtet wird und dass die globalen Scanner für die auszulassenden Zeichen und für die literalen Token verwendet wird.

9.4.9 Startmodus

Es gibt verschiedene Möglichkeiten oder Modi ein TETRA-Programm innerhalb der TETRA-Umgebung auszuführen. Diese Möglichkeiten sind entweder über die Untermenüpunkte des Menüs "Start" auszuführen oder einfacher über die entsprechenden Schalter der Werkzeugleiste oder über die zugeordneten Funktionstasten.

Nächstes Token (F6)
Einzelne Anweisung (F7)
Gesamte Routine (F8)
Start (F9)
Ausführen (F10)
Dateigruppen Transformieren (F11)
Zurücksetzen (Strg+F12)

Wenn einer der ersten fünf Schalter angeklickt wird, wechselt das gesamte Programm in den Debug-Modus. Zugleich wird das gesamte Layout des Programms geändert.



Dieses Bild zeigt das "DebugDefault" Layout. Sie können es an Ihre eigenen Bedürfnisse anpassen.

9.4.10 schrittweise Ausführung des Programms

Die schrittweise Ausführung eines Programms dient dazu Programmzeilen zu lokalisieren an denen eventuelle Fehler (Bugs) auftreten.
Im TextTransformer können die Erkennungen und Aktionen der Knoten des Syntaxbaums einzeln ausgeführt werden. Derjenige Knoten der als nächstes auszuführen ist, wird im Baumdiagramm gelb markiert.
Bei jedem Schritt wird getestet, ob der Text, der dem bisher analysierten Text nachfolgt zu einem der Token passt, die die Anfängermenge des jeweiligen Knotens ausmachen. Ist dies nicht der Fall, ist das Parsen des Textes gescheitert und wird mit einer Fehlermeldung abgebrochen. Wird ein passendes Token gefunden, so wird zum nächsten Knoten fortgeschritten.

Welches der nächste Knoten ist, hängt von der Art ab, wie fortgeschritten wird.

Nächstes Token (F6)

Die Regeln werden bis zu dem nächsten Terminalknoten ausgeführt. Vom aktuellen Knoten des Syntaxbaums bis zu dem des Tokens können mehrere Verzweigungen liegen sowie Knoten, die semantische Aktionen repräsentieren. Die semantischen Aktionen werden dabei sämtlich ausgeführt, falls sie aktiviert sind.

Zum vorherigen Token zurück (UMSCHALT + F6)

Es wird virtuell zum vorherigen Token zurückgegangen. Semantische Aktionen werden hierbei nicht rückgängig gemacht. Beim erneuten Voranschreiten werden solange keine semantischen Aktionen ausgeführt, bis die Position überschritten wird, die bereits einmal erreicht war.

Einzelner Knoten (F7)

Die Unterknoten einer Regel, Option, Wiederholung oder Alternative werden jeweils einzeln ausgeführt. Je nach Position des aktuellen, gelb markierten Knotens gibt es folgende Möglichkeiten:

a) der aktuelle Knoten repräsentiert eine Verzweigung (Option, Wiederholung oder Alternativen), dann wird in den ersten Unterknoten der Verzweigung geschritten.

b) der aktuelle Knoten bezeichnet den Aufruf einer Produktion, dann wird zu deren Darstellung im Baumdiagramm gesprungen.

c) der aktuelle Knoten stellt ein Terminalsymbol dar, ist also ein Blatt der Baumstruktur, dann führt die Ausführung einer einzelnen Anweisung auf den nachfolgenden (unterhalb dargestellten) Knoten, falls es einen solchen gibt. Steht der terminale Knoten am Ende einer Kette so führt die Ausführung einer einzelnen Anweisung auf den Knoten der dem Knoten nachfolgt, der der Kette übergeordnet ist.

Einzelner Knoten zurück (UMSCHALT + F7)

Es wird ein Schritt zurückgegangen. Semantische Aktionen werden hierbei nicht rückgängig gemacht. Beim erneuten Voranschreiten werden solange keine semantischen Aktionen ausgeführt, bis die Position überschritten wird, die bereits einmal erreicht war.

Gesamte Verzweigung (F8)

Die Ausführung einer gesamten Verzweigung unterscheidet sich von der eines einzelnen Knotens in Punkt a) und b). Über die jeweiligen Unterknoten wird hinweg geschritten zum Knoten, der der Verzweigung nachfolgt; d.h. die Unterknoten werden in einem Zug ausgeführt.

9.4.11 schrittweise Ausführung einer Vorausschau

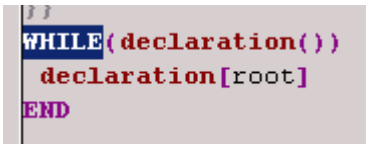
Produktionen können probeweise ausgeführt werden, um in Abhängigkeit vom Erfolg dieselbe oder andere Produktionen auszuführen. Eine solche Vorausschau kann ebenfalls schrittweise getestet werden. Dies ist in beliebiger Staffelung möglich. D.h. der Erfolg einer Vorausschau kann von weiteren Vorausschauen abhängen, die innerhalb der ersten erfolgen.

Ebene der Vorausschau

Die Ebene der Vorausschau wird in einem kleinen Feld innerhalb der Werkzeugleiste angezeigt. Eine leeres Feld oder eine Null bedeuten, dass man sich nicht innerhalb einer Vorausschau befindet, sondern im Hauptparser.

In die Vorausschau (Strg + F7)

Wenn sich der Parser wie in der folgenden Abbildung am Anfang einer IF- oder WHILE-Struktur befindet,



kann mit diesem Schalter in die entsprechende Vorausschau geschritten werden. An anderen Positionen wirkt die Betätigung dieses Schalters wie der Schalter für einen einzelnen Schritt innerhalb einer Vorausschau-Ebene. Wenn das erwartete Token nicht zu der Anfängermenge der Vorausschau-Produktion gehört, wird ebenfalls unmittelbar in der aktuellen Ebene fortgesetzt.

Durch unterschiedliche Markierung der Symbole wird dargestellt,

qualified_id dass es als nächstes getestet wird

qualified_id dass es erfolgreich getestet wurde

ptr_to_member dass der Test nicht erfolgreich war.

Aus der Vorausschau heraus (Strg + F8)

Mit diesem Schalter kann eine Vorausschau verlassen werden. Alle noch anstehenden Schritte innerhalb der aktuellen Ebene werden auf einmal ausgeführt und es wird in der nächst höheren Ebene angehalten.

Anmerkung: die übrigen Schalter und Funktionen des Debuggers sind innerhalb einer Ebene der Vorausschau auf die gleiche Weise zu verwenden wie auf der Ebene des Hauptparsers.

9.4.12 Ausführung des Programms in einem Zug

Bei der Ausführung des Programms in einem Zug wird der Text solange geparst bis er erfolgreich abgearbeitet ist oder bis ein Fehler auftritt. Hierbei gibt es zwei Ausführungsmodi:

Start (F9)

In diesem Modus wird an Haltepunkten, die vor dem Start des Programms gesetzt wurden,

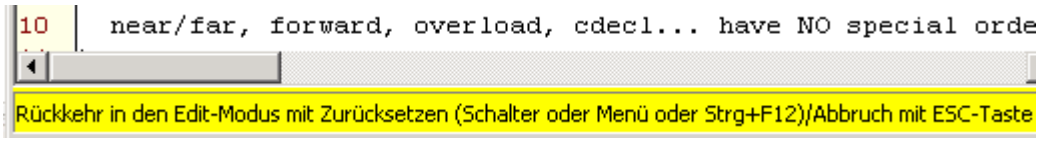
angehalten und die weitere Ausführung kann nach Wunsch schrittweise durchgeführt werden. Im Fall, dass ein Fehler auftritt, gibt es ausführliche Informationen zu den Umständen.

Ausführen (F10)

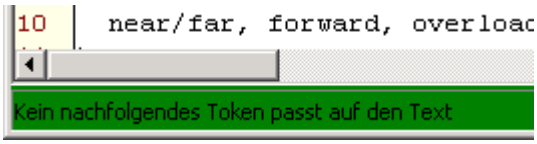
Dieser Modus dient der schnellen Ausführung fertiger Programme. Interne Zwischenschritte die in den anderen Ausführungsmodi notwendig sind werden hier übersprungen. Im Fehlerfall gibt es nur eine allgemeine Nachricht hierüber.

9.4.13 Erfolg feststellen

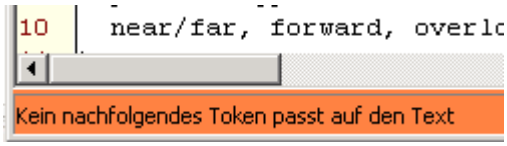
Sobald der Debugger gestartet wird, wird die Status-Zeile gelb gefärbt. Die gelbe Farbe deutet an, dass bisher keinerlei Fehler aufgetreten ist, die Verarbeitung aber noch nicht abgeschlossen ist.



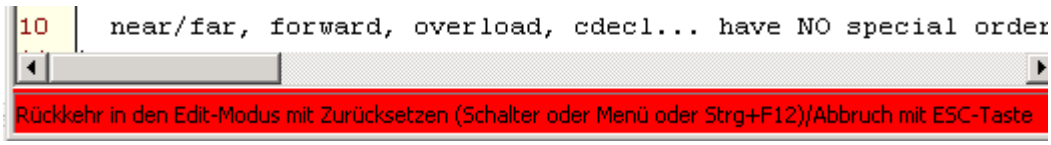
Wenn die Ausführung der Transformation erfolgreich war, so wird dies im Debug-Modus durch eine grün gefärbte Status-Zeile signalisiert.



Wenn vor dem Ende des Textes kein nachfolgendes Token gefunden wird, wird der Parser angehalten und die Status-Zeile wird orange gefärbt.



Es ist weiterhin möglich einzelne Schritte auszuführen, denn semantische Aktionen können noch auf den Abbruch der Erkennungen folgen.
Eine rote Status-Zeile schließlich zeigt an, dass der Parser mit einem Fehler beendet wurde.



Falls der Quelltext nicht vollständig verarbeitet wurde, werden folgende Zeilen an den Zieltext angehängt:

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

```
XX                                                                  XX
XX   An error occurred. The transformation is incomplete!          XX
XX                                                                  XX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Mehr Informationen zum Fehler werden im Log-Fenster angezeigt.

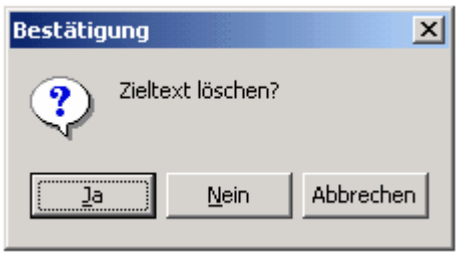
9.4.14 Programm zurücksetzen

Mit dem Befehl

Programm zurücksetzen

kann der Ausführungsmodus von TETRA beendet werden: die expandierten Zweige des Syntaxbaums werden geschlossen, und wenn auf den Namen einer Produktion geklickt wird, wird ihre Definition auf der Produktionenseite angezeigt.

Wurde bereits eine Ausgabe erzeugt, so erlaubt die folgende Dialogbox das Löschen dieser Ausgabe



9.4.15 Erkanntes/erwartetes Token markieren

Wird das Programm schrittweise ausgeführt, so wird neben dem aktuellen Knoten im Syntaxbaum auch die aktuelle Position im Text markiert. Hierfür kann wahlweise das zuletzt erkannte Token angezeigt werden, oder das Token, welches als nächstes erwartet wird. Zwischen diesen beiden Möglichkeiten kann mit einem Schalter gwechselt werden. Ist der Schalter nicht gedrückt,

so bleibt jeweils der zuletzt erkannte Textabschnitt im Eingabetext markiert, bis im Syntaxbaum der nächste Terminal-Knoten durchlaufen wird.
Ist der Schalter gedrückt, so wird nachdem ein Token erkannt wurde, sogleich der Textabschnitt markiert, der dem erwarteten nächsten Token entspricht.

9.4.16 Haltepunkte

Es besteht die Möglichkeit das Programm gezielt nach Erkennung einer bestimmten Textstelle oder vor Ausführung eines bestimmten Knotens anzuhalten, um den weiteren Ablauf schrittweise zu testen. Für die beiden Fälle gibt es spezielle Arten von Haltepunkten:

Text-Haltepunkte
Knoten-Haltepunkte

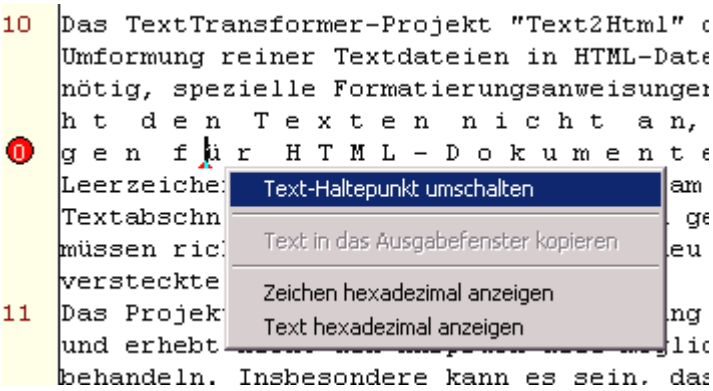
9.4.16.1 Text-Haltepunkte

Nach Erkennung einer bestimmten Textstelle kann das Programm durch einen Text-Haltepunkt gestoppt werden.

Hierzu ist zunächst der Mauscursor auf die gewünschte Position im Eingabetext zu platzieren. Nun kann entweder über das Popup-Menü der rechten Maustaste oder über die Menüleiste unter *Start* der Punkt:

Text-Haltepunkt umschalten

gewählt werden. Nun wird in der gleichen Zeile der Rand des Quelltextfensters rot markiert. Im Eingabeeditor wird ein roter Punkt mit einer weißen Ziffer angezeigt.



Wird in derselben Zeile die Umschalt-Operation ein zweites Mal betätigt, so wird der zuvor gesetzte Haltepunkt wieder entfernt.

Ein Haltepunkt am Anfang einer Zeile kann auch einfach durch einen Mausklick auf den Seitenrand gesetzt und entfernt werden.

Nur im Editor:
lassen sich Haltepunkte mit einer bestimmten Nummer setzen, indem Strg + Umschalt + Ziffer gleichzeitig gedrückt werden. Mit der gleichen Tastenkombination kann ein Haltepunkt entfernt werden, wenn sich der Eingabe-Cursor an einer beliebigen Stelle der Zeile befindet.

Insgesamt lassen sich zehn Haltepunkte setzen (0 - 9). Im Editor können die Haltepunkte

angesprungen werden, indem die Tasten Strg + Ziffer gedrückt werden.

Wird der Text editiert, nachdem Haltepunkte gesetzt wurden, so kommt es zu Verschiebungen derjenigen Haltepunkte, die hinter dem editierten Textabschnitt liegen. Sie sollten daher entfernt werden.

Der Menüpunkt unter *Start*:

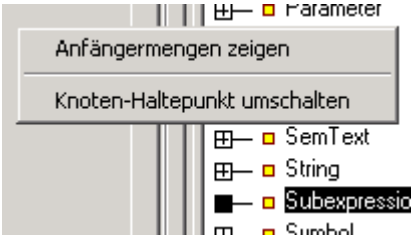
Text-Haltepunkte löschen

entfernt sämtliche im Eingabetext gesetzten Haltepunkte.

9.4.16.2 Knoten-Haltepunkte

Vor der Ausführung eines bestimmten Knotens kann das Programm durch einen Knoten-Haltepunkt gestoppt werden.

Der Knoten, an dem ein solcher Haltepunkt gesetzt werden soll, wird zunächst im Syntaxbaum mit der linken Maustaste ausgewählt. Ist dies geschehen erscheint beim Drücken der rechten Maustaste ein lokales Popup-Menü in dem der Eintrag: "Knoten-Haltepunkt umschalten" ausgewählt werden kann.



Nachdem der Haltepunkt so gesetzt wurde, wird der entsprechende Knoten im Baumdiagramm rot markiert dargestellt. Nochmaliges Umschalten des Knoten-Haltepunkts entfernt diesen wieder.

Bei jedem Parsen der Regeln (ersichtlich an der Fortschrittsanzeige) werden sämtliche Knoten-Haltepunkte entfernt.

Knoten-Haltepunkte können nur in Regeln des Hauptparsers gesetzt werden.
Vorausschau-Produktionen und Unterparser sind Interpreter-Aufrufe. Um in ihnen Knoten-Haltepunkte zu setzen, müssen sie zum Test als Startregel gesetzt werden.

9.4.17 Erkanntes Token

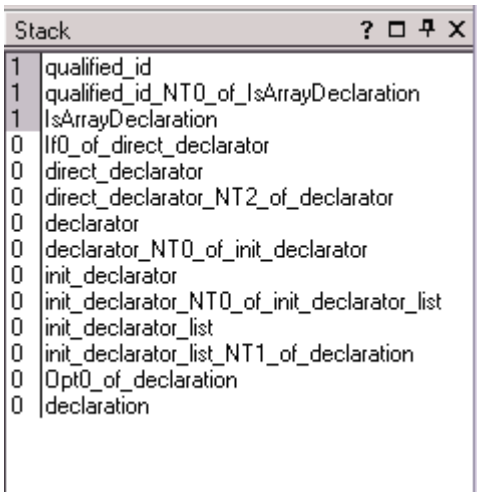
Ein eigenes andockbares Fenster bildet eine kleine Box mit Informationen über das zuletzt erkannte und das gefundene Token.

	Name	No
erkannt		
gefunden	"init"	14
gefunden		

Bei der schrittweisen Ausführung eines TETRA-Programms werden in einer kleinen Tabelle jeweils in der ersten Zeile das zuletzt erkannte Token angezeigt und in der zweiten Zeile, dasjenige Token das als nächstes gefunden wurde. Sobald der Knoten des erwarteten Terminalsymbols durchschritten wird, wird es akzeptiert und somit zum erkannten Token. Zugleich wird das nächste Token ermittelt. Ein Sonderfall sind die SKIP-Knoten. Sobald eine SKIP-Alternative gewählt wurde, ist auch das dem SKIP-Knoten nachfolgende Token bekannt. Es wird in der dritten Zeile der Tabelle angezeigt.

Die Combobox oberhalb der Token-Tabelle enthält eine Liste sämtlicher Token, von denen die geparsten Regeln abhängen. Für jedes Token wird jeweils der Name, eine von TETRA vergebene Symbolnummer (in Klammern) und der Text des regulären Ausdrucks angegeben.

9.4.18 Stack-Fenster



Wird bei der Ausführung eines TETRA-Programms von einer Produktion in eine andere verzweigt oder wird ein optionaler Zweig innerhalb einer Produktion gewählt, so wird der Name des übergeordneten Knotens in der obersten Zeile des Stackfensters eingetragen. Die bisherigen Einträge dieses Fensters rücken dabei eine Zeile nach unten. Im Stackfenster wird so die Reihe der übergeordneten Knoten dargestellt, die den "Weg" kennzeichnen, auf dem zum aktuellen Knoten gelangt wurde.

Die vorangestellte Nummer bezeichnet die Ebene der Vorausschau.

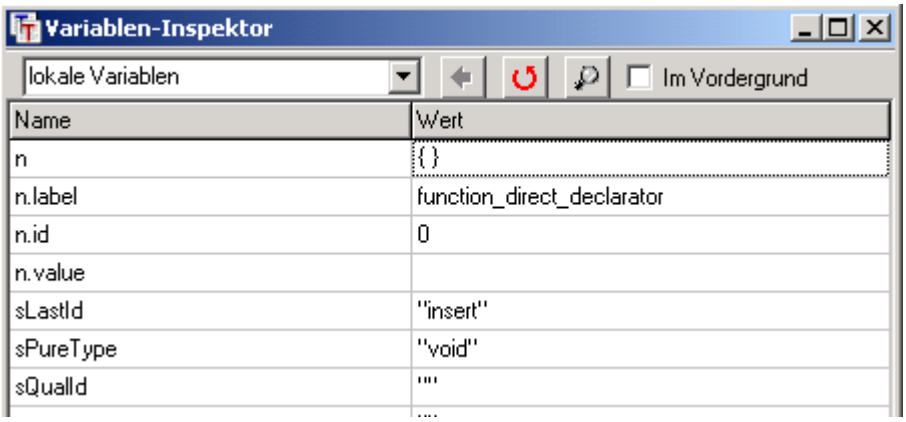
Bei Mausklick auf eine Zeile des Stackfensters wird im Syntaxbaum der jeweils zugehörige Knoten angezeigt und markiert und im Eingabetext wird der Textabschnitt markiert, der durch die zugehörige Struktur erkannt wurde.

9.4.19 Variablen-Inspektor

Mit dem Variablen-Inspektor können während des Debuggens die Inhalte von Variablen des aktuellen Sichtbarkeitsbereichs betrachtet werden.

Das Fenster des Variablen-Inspektors kann im Debug-Modus (nicht in einer Vorausschau, s.u.) über den Schalter

oder den entsprechenden Menüpunkt im Start-Menü aufgerufen werden.



In der Auswahlbox links oben kann entweder der Name einer ausgewählten Variablen eingegeben werden oder einer der fünf vorgegebenen Einträge ausgewählt werden:

Klassen-Variablen

Nach Auswahl des Eintrags *Klassen-Variablen* werden die Variablen angezeigt, die auf der Element-Seite definiert sind. (Der Parser selbst wird als *this* bzw. (**this*) bezeichnet.)

lokale Variablen

Nach Auswahl des Eintrags *lokale Variablen* werden die Werte aller Variablen angezeigt, die im aktuellen Variablen-Bereich liegen: das sind die an die aktuelle Produktion übergebenen Variablen sowie die Variablen, die in ihr lokal deklariert sind.

xState (Parser-Zustand)

Nach Auswahl des Eintrags *xState* werden alle Elemente der Parser-Zustands-Variablen angezeigt. Sämtliche Unterausdrücke des zuletzt gefundenen Tokens werden aufgelistet.

Plugin Variablen

Nach Auswahl des Eintrags *Plugin Variablen* werden die Variablen des Plugins angezeigt, d.h. insbesondere Quell- und Zielangaben und Angaben zum Einrückungs- und Textbereich-Stack angezeigt

DOM

Wenn im Programm ein DOMDocument erzeugt wurde, kann es hier unter angezeigt werden.

Schalter der Werkzeugleiste

Nach oben in der Hierarchie 

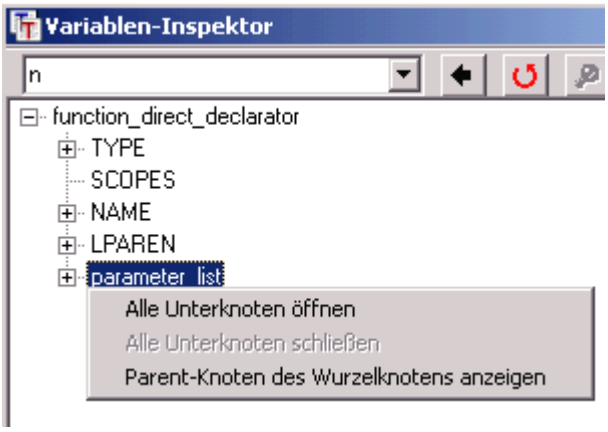
Mit dem Zurück-Schalter kommt man eine Ebene höher in der Hierarchie der Klassenelemente oder schließlich zur Ansicht aller Variablen eines Sichtbarkeitsbereichs (s.o.).

Aktualisieren 

Nach dem Öffnen des Variablen-Inspektors muss der *Aktualisieren*-Schalter betätigt werden, um den Wert einer eventuell bereits ausgewählten Variablen zu aktualisieren.

Details = 
Auswahl einer einzelnen Variablen durch Doppelklick

Sind mehre Variablen zugleich angezeigt, können über den *Details*-Schalter oder durch Doppelklick mit der linken Maustaste auf der Werte-Seite einzelne Variablen ausgewählt werden, um sie vollständig anzuzeigen. So können auch längere Textinhalte, die Elemente von Containern oder - wie in folgendem Beispiel - ganze Baumstrukturen eingesehen werden.



In einer Baumansicht können Knoten mit der rechten Maustaste ausgewählt werden um über ein Popup-Menü alle Unterknoten aufzuklappen. Über dieses Menü ist es ebenfalls möglich Zweige wieder einzuklappen und man kann die Wurzel des Baums zu dessen Parent-Knoten verschieben.

Im Vordergrund

Wird das Häkchen in der Checkbox *Im Vordergrund* gesetzt, so bleibt der Variablen-Inspektor während der gesamten Debugging-Sitzung sichtbar im Vordergrund. Nach jedem Debug-Schritt wird der Inhalt der ausgewählten Variable automatisch aktualisiert. Ist das Häkchen nicht gesetzt unterbleibt die automatische Aktualisierung und der Inspektor wird bei jeder Aktion vom Bildschirm gelöscht.
Wird der Inspektor bei gesetztem Hächen geschlossen, so wird die Vordergrunds-Option deaktiviert.

Handelt es sich um die Anzeige einer komplexe Variable, wie z.B. der Parserzustand xState, werden die die Werte der Klassen-Elemente der Variablen angezeigt. In einigen Fällen werden neben den eigentlichen Werten der Variablen auch Eigenschaften angezeigt. Da Container (mstrstr, vstr) sehr viele Werte enhalten können, wird bei ihnen lediglich die Anzahl der Elemente angegeben.

Beim Debuggen einer Vorausschau wird der Variablen-Inspektor nicht angezeigt, weil während einer Vorausschau keine semantischen Aktionen ausgeführt werden.

9.4.20 Zur aktuellen Position

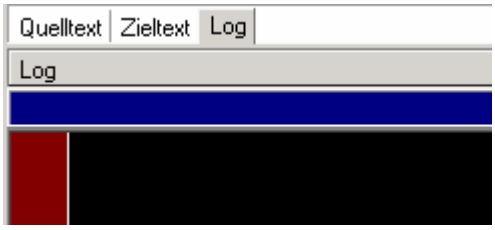
Durch diese Funktion kann der Zustand wiederhergestellt werden, der nach dem letzten Schritt im Debugger bestand: das letzte erkannte Token wird im Text markiert und der aktuelle Knoten im Syntaxbaum wird markiert.

9.4.21 Info-Box

Über *Menü->Start->Letzte Meldung zeigen* oder den Schalter

lässt sich die letzte Information erneut aufrufen, die zuvor schon einmal angezeigt wurde: z.B. die Tokenmengen zu einem Baumknoten.

9.4.22 Log-Fenster



Im Log-Fenster werden Meta-Informationen zum Ablauf eines Programms angezeigt. Hierzu gehören Erfolgs- und Fehlermeldungen und auch Ausgaben, die im Interpreter-Code explizit noch log geschrieben werden.

Wenn das Log-Fenster Informationen enthält, wird dies durch ein rotes Kästchen signalisiert.

9.5 Transformation von Dateigruppen

Es gibt zwei Möglichkeiten Dateien stapelweise zu transformieren:

1. interaktiv direkt aus dem TextTransformer heraus oder
2. mit Hilfe eines zusätzlichen Kommandozeilen-Programms: *tetra_cl*.

9.5.1 Transformations-Manager

Der Transformations-Manager ist eine Dialogbox zum Transformieren ganzer Verzeichnisse oder anderer Gruppen von Dateien.
Der Transformations-Manager ist entweder über den Menüpunkt *Dateigruppen transformieren* des *Start*-Menüs zu erreichen oder über den entsprechenden Schalter in der Werkzeugleiste

Bevor der Dialog geöffnet wird, muss zunächst die aktuelle Startregel - die dann für die Transformation verwendet wird - kompiliert sein. Falls dies noch nicht erfolgt ist, geschieht es automatisch, beim Öffnen des Managers.

Der Schalter der Werkzeugleiste des Managers zum Ausführen der Transformationen ist zunächst deaktiviert, da noch keine Quelldateien für die Transformation ausgewählt sind. Erst, wenn dies geschehen ist, und Optionen zur Transformation wunschgemäß sind, kann die Transformation

gestartet werden. Zuvor kann noch die Liste der Dateien überprüft werden, die bei den Transformationen erzeugt werden. Für jeden dieser Schritte gibt es eine eigene Seite im Transformations-Manager:

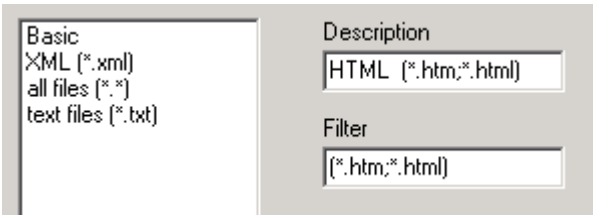
1. Quelldateien
2. Tranformations-Optionen
3. Vorausschau auf die Liste der Zieldateien
4. Resultate

Die Einstellungen inklusive der ausgewählten Verzeichnisse und Dateien können als Management gespeichert und bei Bedarf neu geladen werden.

9.5.1.1 Neuen Filter definieren

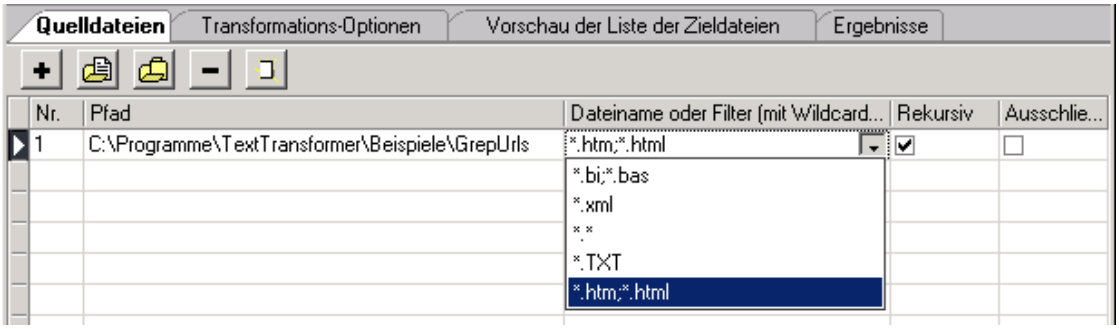
Die Auswahl der Quelldateien wird erleichtert, wenn zuvor bereits Datei-Filter für häufig benötigte Dateitypen definiert sind.

Über das Menü des Transformations-Managers ***Neuer Datei-Filter*** kann eine neue Maske für die zu transformierenden Dateien definiert werden. Der Dialog, der bei dieser Aktion erscheint, ist der gleiche, der auch in den Umgebungsoptionen des TextTransformers angezeigt wird.



9.5.1.2 Quelldateien auswählen

Die Dateien, die transformiert werden sollen, werden auf der ersten Seite des Transformations-Managers ausgewählt und in einer Tabelle angezeigt.



Die Seite hat eine eigene Werkzeugleiste mit den Schaltern:


Eine leere Spalte in die Tabelle einfügen

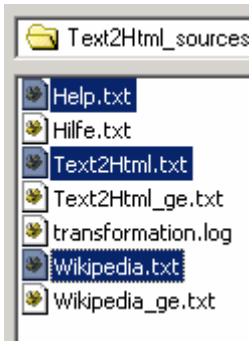

Einzelne Quelldatei auswählen


Ganzes Quellverzeichnis auswählen

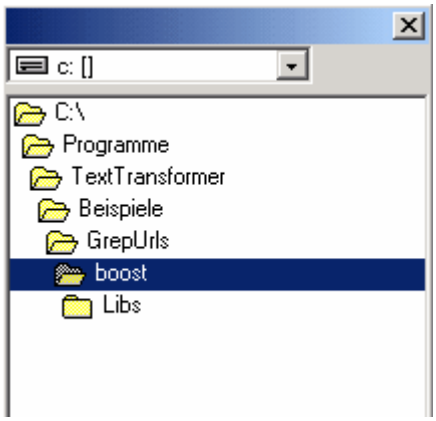

Zeile löschen


Gesamte Tabelle löschen

Die Auswahl einer Datei bzw. eines Verzeichnisses erfolgt jeweils mit einer entsprechenden Auswahlbox. In der Auswahlbox für Dateien können auch mehrere Dateien auf einmal selektiert werden.



Nach der Bestätigung der Wahl wird für jede Datei bzw. jedes Verzeichnis eine neue Zeile in der Tabelle unterhalb der Werkzeugleiste erzeugt.



In der Tabelle gibt es fünf Spalten:

Nr.

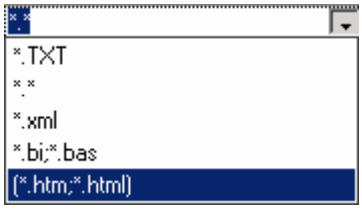
ein einfacher Zähler

Pfad

Der absolute Pfad einer Datei oder eines Ordners.

Dateiname oder Filter

Für ausgewählte Dateien ist hier der Dateiname samt Erweiterung zu sehen.
Für Ordner kann hier ein Filter spezifiziert werden. Bereits definierte Filter können in dieser Spalte aus einer Combo-Box mit der Maus ausgewählt werden,



es können aber auch beliebige andere Filter direkt in das Feld geschrieben werden. Wird z.B. der Filter "*.txt" ausgewählt, so werden nur die Dateien des betreffenden Ordners transformier, die die Erweiterung "txt" haben.

Rekursiv

Die Check-Box in diesem Feld kann nur für Ordner aktiviert werden. Ist sie aktiviert, so werden auch alle Dateien in den Unterordnern des angezeigten Verzeichnisses transformiert.

Ausschließen

Normaler Weise ist die Check-Box dieses Feldes deaktiviert. Es kann aber sein, dass Sie bei der Transformation eines Ordners einige Dateien oder Ordner ausnehmen wollen. Dies ist möglich, indem Sie für diese Ausnahmen eigene Zeilen in der Tabelle erzeugen und per Maus das Hächen in der *Ausschließen*-Check-Box setzen.

Extra

Extra-Parameter pro Datei oder Dateigruppe können im letzten Feld gesetzt werden.

9.5.1.3 Transformations-Optionen

Über den Config-Schalter lässt sich ein kleiner Editor öffnen, in den man Konfigurationsparameter für die Transformationen schreiben kann.

Grundsätzlich gibt es zwei Weisen im Transformations-Manager, die Quelldateien zu transformieren:

1. **N:N**: jede Quelldatei wird in eine Zieldatei abgebildet
2. **N:1**:. sämtliche Dateien werden in nur eine Zieldatei abgebildet.

Je nachdem, ob eine N:N oder eine N:1 Transformation beabsichtigt ist, gibt es verschiedene Sets von Einstellungen.

Log-Datei

Ungeachtet der Art der Transformation kann aber eine Log-Datei bestimmt werden, in der die Aktionen und Meldungen der Transformationen protokolliert werden. Die Informationen die in eine solche Log-Datei geschrieben werden sind u.U. umfassender als die Meldungen auf der Resultate-Seite.

9.5.1.3.1 N:N Transformation

N:N: jede Quelldatei wird in eine Zieldatei abgebildet

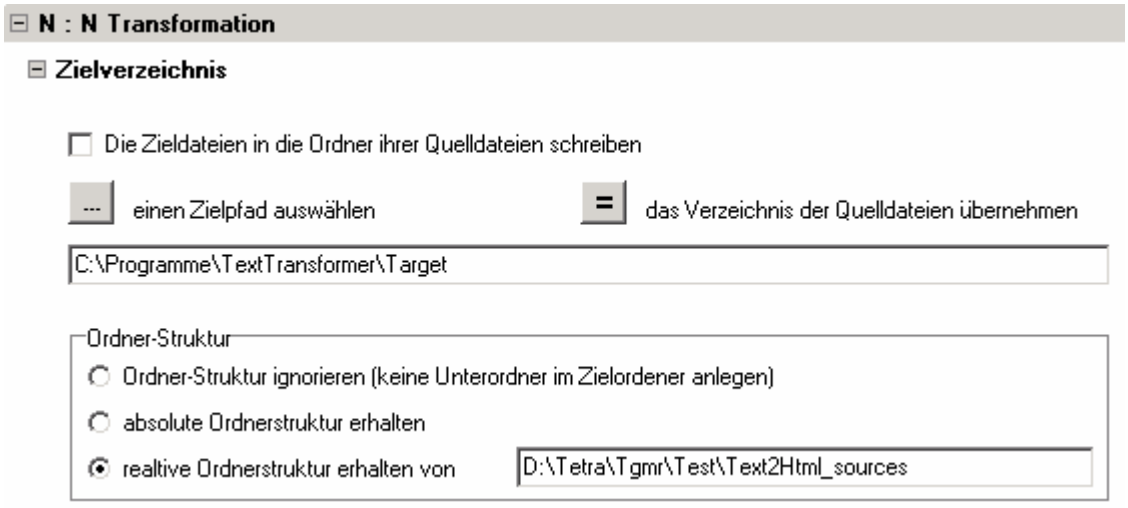
Dies ist der Standardfall im Transformations-Manager, bei dem ein Quelltext in einen Zieltext übersetzt wird. Es wird die gleiche Anzahl von Zieldateien erzeugt, wie Quelldateien vorhanden sind. Es ist prinzipiell erlaubt, die ursprünglichen Dateien durch ihre Übersetzung zu überschreiben. Jedoch sollte für eine entsprechende Sicherung gesorgt werden. Der TextTransformer kann optional Sicherungskopien der Quelldateien in einem gewählten Backup-Verzeichnis anlegen, bevor die Transformierung beginnt.

9.5.1.3.1.1 Zielverzeichnis auswählen

Es ist sowohl möglich, die Quelldateien überschreiben zu lassen, als auch die transformierten Texte in einem anderen Verzeichnis zu erzeugen.

Die Zieldateien in den Ordner ihrer Quelldateien schreiben

Überschrieben werden die Quelldateien dann, wenn die Option "Die Zieldateien in den Ordner ihrer Quelldateien schreiben" aktiviert ist und kein besonderes Muster für die Zieldateien bestimmt wird. Es braucht weder ein gesondertes Zielverzeichnis ausgewählt zu werden, noch eine Ordner.Struktur für die Ziele gewählt zu werden. Die entsprechenden Felder sind deshalb dann inaktiv.



Einen neuen Ausgabe-Ordner bestimmen

Wenn das Kästchen "Die Zieldateien in den Ordner ihrer Quelldateien schreiben" inaktiv ist, werden die Felder zur Bestimmung der Zieldordner aktiv.

Nach dem ersten Öffnen des Transformations-Managers ist das Zielverzeichnis auf das in den Umgebungsoptionen angegebene Verzeichnis gesetzt. Es kann aber im Dialog temporär verändert werden.

Mit dem Schalters

wird eine Dialogbox zur Auswahl eines anderen Zielverzeichnisses geöffnet.

Der Schalter

kann ein ein Hilfsmittel sein, um schneller zum neuen Zielverzeichnis zu navigieren.:

Ordner-Struktur

Wenn alle Quelldateien den gleichen Pfad haben, spielen die folgenden Optionen zur Ordner-Struktur keine Rolle. Stammen sie hingegen aus verschiedenen Verzeichnissen, so gibt es mehrere Möglichkeiten für die Konstruktion der Pfade der Zieldateien:

Ordner-Struktur ignorieren

bedeutet, dass alle Zieldateien den gleichen Pfad des Zielverzeichnisses erhalten.

Beispiel:

Ist der Zielordner: C:\Targets

so ergibt die Transformation der Dateien

C:\Programme\TextTransformer\Source.cpp
C:\Programme\TextTransformer\Sources\Source.txt

die folgenden Zieldateien:

C:\Targets\Source.cpp
C:\Targets\Source.txt

Sollte sich unter den Quelldateien auch "C:\Source.txt" befinden, so kommt es zu einer Überschneidung der Zieldateien und es erfolgt eine entsprechende Fehlermeldung.

Ordner-Struktur absolut bewahren

Beispiel:

Ist der Zielordner: C:\Targets

so ergibt die Transformation der Dateien

C:\Programme\TextTransformer\Source.cpp
C:\Programme\TextTransformer\Sources\Source.txt
C:\Source.txt

die folgenden Zieldateien:

C:\Targets\Programme\TextTransformer\Source.cpp
C:\Targets\Programme\TextTransformer\Sources\Source.txt
C:\Targets\Source.txt

Die Datei "C:\Source.txt" bereitet hier keine Probleme.

Ordner-Struktur relativ bewahren

Beispiel:

Ist der Zielordner: C:\Targets

so ergibt die Transformation der Dateien

C:\Programme\TextTransformer\Source.cpp
C:\Programme\TextTransformer\Sources\Source.txt

die folgenden Zieldateien:

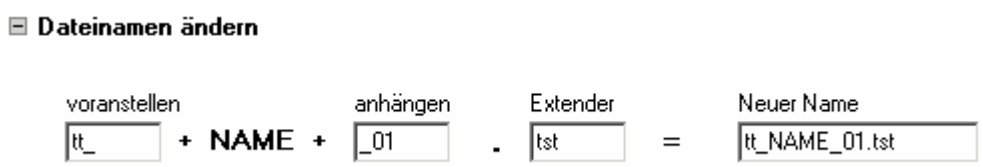
C:\Targets\Source.cpp
C:\Targets\Sources\Source.txt

Sollte sich unter den Quelldateien auch "C:\Source.txt" befinden, so muss als Startverzeichnis für die relative Ordner-Struktur "C:" gewählt werden und man erhält dasselbe Ergebnis wie bei der Bewahrung der absoluten Ordner-Struktur.

9.5.1.3.1.2 Muster für die Zieldateien setzen

Die Namen der Quelldateien können bei der Transformation im Transformations-Manager verändert werden, indem sie mit einem Pre- oder Postfix versehen werden, oder indem ihre Erweiterung verändert wird. Das Muster für die Bildung der Namen der Zieldateien wird durch die Eingabe in die entsprechenden Felder bestimmt.

Beispiel:

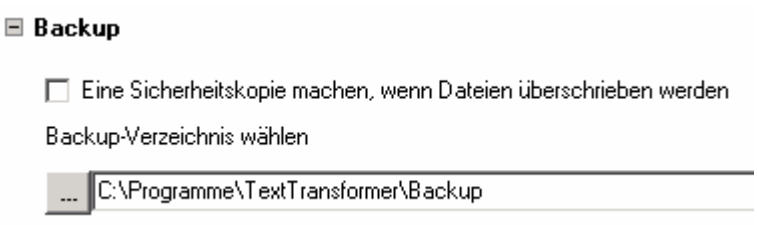


ändert die Dateinamen in folgender Weise:

test.dat	->	tt_test_01.tst
Source.txt	->	tt_Source_01.tst

9.5.1.3.1.3 Backup

Wenn durch die Transformation bestehende Dateien überschrieben werden, so ist es ratsam vorher ein Backup von ihnen zu machen. In den Optionen des Transformations-Managers gibt es daher die Möglichkeit, einen Ordner auszuwählen, in den die ursprünglichen Dateien vor der Transformation kopiert werden.



Auch, wenn die Backup-Einstellung aktiviert ist, wird ein Backup nur dann gemacht, wenn tatsächlich mindestens eine Datei überschrieben werden wird. In diesem Fall werden dann sämtliche

Quelldateien gesichert.

Bevor eine Transformation beginnt, wird geprüft, ob gemäß den Einstellungen Quell-Dateien überschrieben werden. Ist dies der Fall, erfolgt eine Warnung, die es noch erlaubt, die originalen Texte durch ein Backup zu sichern.

Mit der Roll-back-Funktion können die Backup-Dateien zurückkopiert werden, solange die Einstellungen im Transformations-Manager nicht verändert wurden.

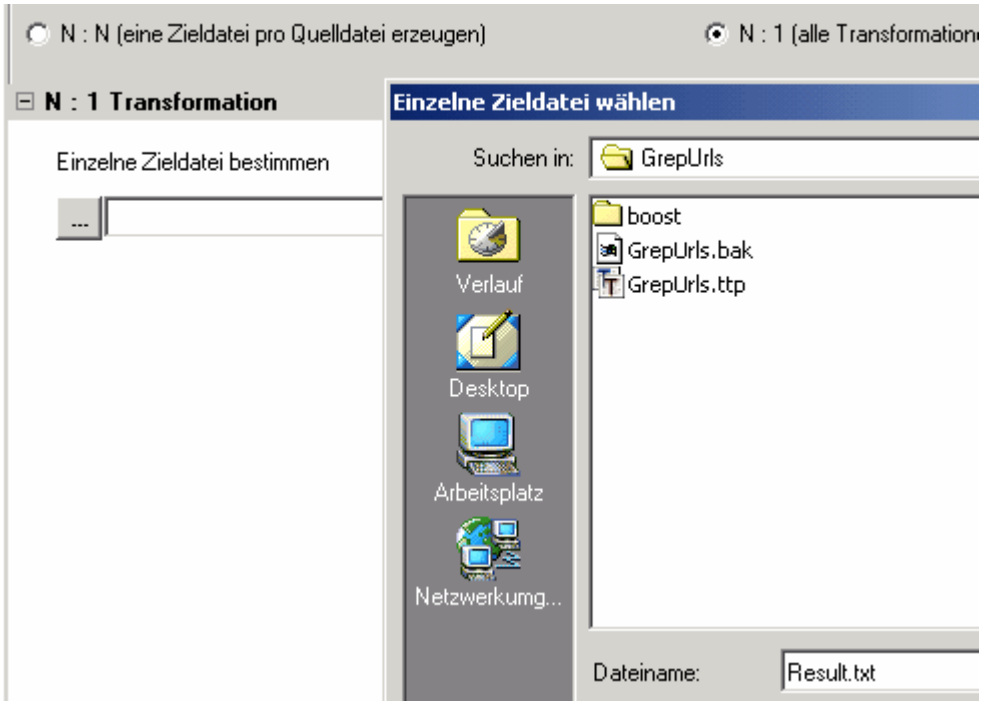
Anmerkung: Falls Pfad und Name einer Zieldatei mit denen der Quelldatei identisch ist, wird das Ergebnis der Transformation zunächst in eine **temporäre Datei** geschrieben, die bei erfolgreicher Transformation dann in den ursprünglichen Namen umbenannt wird. Tritt bei der Transformation ein Fehler auf, so findet die Umbenennung nicht statt und die temporäre Datei verbleibt im Zielverzeichnis. Sie kann in einem Editor betrachtet werden, um festzustellen wo der Fehler auftrat. Die Namen der temporären Dateien werden zusammengesetzt aus *temporary*, eventuell gefolgt von einer Zahl und der *tmp* Erweiterung.

9.5.1.3.2 N:1: Transformation

N:1:. sämtliche Dateien werden in nur eine Zieldatei abgebildet.

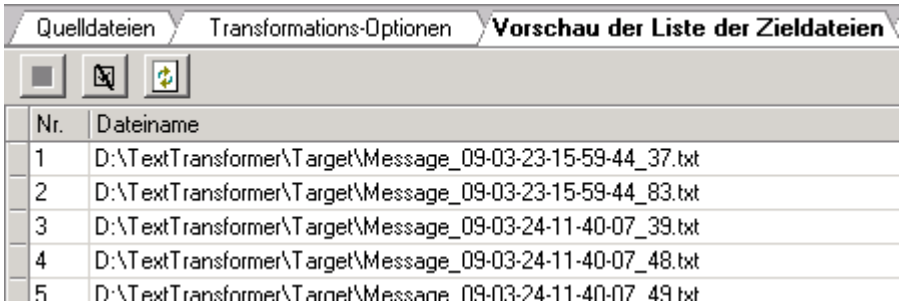
Sollen aus einer Vielzahl von Dateien im Transformations-Manager Informationen extrahiert und in einer neuen Datei zusammengefasst werden, so besteht einen N:1 Beziehung von Quelle und Ziel. In diesem Fall muss die Zieldatei von allen Quelldateien verschieden gewählt werden. Ein Backup der Quelldateien ist daher nicht erforderlich.

Die Zieldatei kann über einen Auswahldialog geöffnet werden. Es ist möglich den Dialog nur zur Auswahl des Verzeichnisses zu verwenden und einen noch nicht existierenden Dateinamen im Dialog einzutragen.



9.5.1.4 Vorschau der Zieldateien

Die Liste der zu erzeugenden Dateien wird auf der dritten Tab-Seite des Transformations-Managers angezeigt.



Im Falle einer N:1 Transformation zeigen die Zeilen der Tabelle, welche Quelldateien zur Zieldatei beitragen können.

Einzelne Dateien ausschließen

Wie schon die Tabelle der Quell-Dateien so enthält auch die Tabelle der erwarteten Ziel-Dateien ein Feld mit einer Check-Box zum ausschließen einzelner Dateien. Hier besteht nun die Möglichkeit einzelne Dateien auszuschließen, deren Quell-Datei in einem ausgewählten Quell-Verzeichnis liegt.

Anmerkung:
Beim Schreiben eines Managements werden die zu den ausgeschlossenen Zieldateien gehörenden Quelldateien ausgeschlossen. Wenn das Management dann geladen wird, erscheinen dann also

diese Dateien in der Quelltext-Tabelle und nicht mehr in der Zieltext-Tabelle.

Erfolgreich transformierte Dateien ausschließen

Wurde die Transformation der zum Management gehörenden Dateien aktuell bereits einmal ausgeführt, so lassen sich mit dem Schalter in der Werkzeugleiste all diejenigen Dateien ausschließen, die bereits erfolgreich transformiert wurden. So ist es möglich, ein korrigiertes Projekt allein auf die Dateien anzuwenden, die bisher nicht umgewandelt werden konnten.

Alle Dateien einschließen

Mit dem Schalter in der Werkzeugleiste kann der Ausschluss aller Dateien aufgehoben werden.

Neu einlesen

Mit dem Schalter kann die Dateiliste aktualisiert werden.

9.5.1.5 Transformation starten

Die Transformation der ausgewählten Dateien wird im Transformations-Manager über den Menüpunkt Transformation starten oder den Schalter

in der Haupt-Werkzeugleiste gestartet.

Mit dem Start wird automatisch auf die Resultate-Seite gewechselt.

9.5.1.6 Resultate

Die Zeilen der Tabelle auf der Resultate-Seite der Transformations-Managers enthalten Meldungen, die während der Transformation von Dateien entstehen. Meldungen werden vom Transformations-Manager automatisch erzeugt. Es werden aber auch Meldungen angezeigt, die vom Benutzer programmiert wurden. Jede Meldung wird sofort nach ihrem Entstehen in eine neue Zeile der Tabelle geschrieben. Die wachsende Zeilenzahl der Tabelle ist so zugleich eine Fortschrittsanzeige für die Transformationen.

S...	Datum	Zeit	Meldung
	10.07.2006	20:33:24	Starte N:N Transformation
	10.07.2006	20:33:24	zu überspringender Text (SKIP) erwa
	10.07.2006	20:33:24	D:\Tetra\Projects\TextTransformer\
	10.07.2006	20:33:26	erfolgreich transformiert : D:\Tetra\P
	10.07.2006	20:33:26	D:\Tetra\Projects\TextTransformer\
	10.07.2006	20:33:26	erfolgreich transformiert : D:\Tetra\P
	10.07.2006	20:33:26	erfolgreich transformiert : D:\Tetra\P
	10.07.2006	20:33:26	erfolgreich transformiert : D:\Tetra\P
	10.07.2006	20:33:26	Letzte Transformation beendet

In der ersten Spalte wird der Status der Meldung als Farbwert angezeigt:

Farbe	Status	benutzerdefiniert durch
	neue Quelldatei	-
	neutrale Information	AddMessage
	Erfolgsmeldung	–
	Warnhinweis	AddWarning
	Fehlermeldung	AddError

Aus der Ergebnisliste kann ein Report erstellt werden.

9.5.1.6.1 Report

Aus der Liste der Resultate im Transformations-Manager kann ein Report geschrieben werden. Dazu ist über den Menüpunkt "Datei->Resultate schreiben als..." oder direkt über den Schalter auf der Resultateseite

eine vorhandene Datei auszuwählen oder es wird ein Verzeichnis gewählt und der Name für eine neue Datei in das entsprechende Feld geschrieben..

Die Informationen, die in den Report aufgenommen werden, können vom Benutzer bestimmt werden. Nur Angaben für die ein Häkchen im Dialog gesetzt ist, werden ausgegeben.

Eine Nummerierung der Quelldateien empfiehlt sich, weil es manchmal mehrere Meldungen zu einer Datei gibt.
Den Farbwerten der ersten Spalte der Resultatetabelle entsprechen hier die verschiedenen Kästchen zum Status. Nur wenn das Häkchen gesetzt ist, werden Zeilen mit dem entsprechenden Status ausgegeben.

9.5.1.7 Korrekturen vornehmen

Nachdem die Transformation beendet ist, kann man sich zunächst die transformierten **Dateien anschauen** und bei Bedarf Änderungen am Projekt vornehmen. Nach einem **Doppelklick** auf eine Zeile der Resultat-Tabelle oder über ein Popup-Menü wird der Managerdialog geschlossen und die in der Zeile genannte Quelldatei und - falls vorhanden - die zugeordnete Zieldatei werden in das Eingabe- bzw. Ausgabefenster der IDE geladen.

Es ist möglich, das Projekt zu verbessern, dann den Transformations-Manager mit den gleichen Einstellungen erneut zu öffnen und eventuell die gesamten Transformationen zu wiederholen.

9.5.1.8 Roll back

Hat man das Projekt verbessert, kann man es nochmals mit den gleichen Quell- und Zieldateien ausführen. Hatte man die ursprünglichen Dateien überschrieben, so kann vor der erneuten Transformierung mittels des Schalters:

das Backup an den originalen Ort zurück kopiert werden.

Vorsicht! Wurden die Quelldateien überschrieben und wurde vor einer erneuten Ausführung das Backup nicht zurück kopiert, so wird das Backup selbst durch die schon überschriebenen Dateien überschrieben. **Es ist stets sicherer für die Zieldateien ein anderes als das Quellverzeichnis zu wählen.**

9.5.1.9 Management

Als *Management* wird hier die Summe der Einstellungen des Transformations-Managers bezeichnet. Ein Management ist ein Transformations-Manager-Projekt. Um einer Verwechslung mit einem TextTransformer-Projekt vorzubeugen, wird es als Management bezeichnet.

Über das Menü: ***Management speichern als***, kann ein Management gespeichert werden.

Über das Menü: ***Management öffnen***, kann ein Management erneut geladen werden. Auf die zuletzt verwendeten Managements kann durch die History-Liste im Menü schnell zugegriffen werden.

Ein Management kann auch zur Steuerung des Kommandozeilen-Werkzeugs Tetra_cl eingesetzt werden.

Managements werden mit der Erweiterung "ttm" abgespeichert. Sie werden mit dem Projekt FileList.ttp geparst.

Die Syntax für ein Management wurde möglichst knapp und einfach gestaltet, damit es auch von Hand geschrieben werden kann. Im Extremfall besteht ein Management aus nur einer Dateiangabe. Felder zu inaktiven Optionen werden in einem Management nicht gespeichert, auch, wenn sie im Transformationsmanager lesbar sind. Beim Parsen eines Managements wird anhand der Existenz von Werten entschieden, ob die Optionen für die sie benötigt werden aktiv sind. So entscheidet z.B das Vorhandensein eines Zielpfades darüber, ob Dateien überschrieben werden oder nicht.

Ein Beispiel für ein Management ist im Verzeichnis des GrepUrls-Beispiels zu finden:

```
single_target = C:\Programme\TextTransformer\Beispiele\GrepUrls\Result.txt
log_file = C:\Programme\TextTransformer\Log\transformation.log
+ r C:\Programme\TextTransformer\Beispiele\GrepUrls\*.htm;*.html
```

9.5.2 Kommandozeilenprogramm

Vom TETRA-Programm gibt es die Kommandozeilenversion **tetra_cl**, die die automatisierte Transformation von Dateien erlaubt. Ab Version 1.2.2 kann das Kommandozeilen-Werkzeug optional bei der Installation des TextTransformers mit installiert werden. Es befindet sich dann im bin-Ordner des Programmverzeichnisses.

Beim Aufruf von **tetra_cl** müssen Parameter angegeben werden, denen das Programm entnimmt, welche Dateien mit welchem TETRA-Projekt auf welche Weise transformiert werden sollen.

9.5.2.1 Parameter

Tetra_cl lässt sich entweder über ein Management steuern, das mit dem Transformations-Manager erzeugt wurde, oder durch Parameter für die Quell- und Ziel-Dateien.
Im ersten Fall hat der Aufruf die Form:

```
tetra_cl -p PROJECT -m MANAGEMENT [-a]
```

im zweiten Fall ist er:

```
tetra_cl -p PROJECT -s SOURCE [-t TARGET] [-b BACKUP] [-c CONFIGURE] [-x EXTRA] [-a] [-r]
```

Die einzelnen Parameter sind stets mit einem Bindestrich gekennzeichnet, auf den unmittelbar ein

Buchstabe folgt. Nach einem Leerzeichen muss bei einigen Parametern ein Text folgen. Die Ausdrücke in eckigen Klammern sind optional.
Pfadangaben, die Leerzeichen enthalten müssen in Anführungszeichen gesetzt werden, z.B.: "C:\Dokumente und Einstellungen\admin\Eigene Dateien\Ziel"

Parameter	Bedeutung	Beispiele
-p PROJECT	TETRA-Projekt	exchange.ttp
-m MANAGEMENT	Transformations-Manager Projekt	MyWebSite.ttm
-s SOURCE	Quelldatei(en)	C:\dir*.txt
-t TARGET	Zieldatei oder Zielverzeichnis	C:\dir2\target.txt
-b BACKUP	Backup-Verzeichnis	C:\Backup
-c CONFIGURE	Konfigurationsparameter	"\"C:\\boost\", \"C:\\mylib\""
-x EXTRA	Extra-parameter	alternative_rule
-a ASK	Abfrage bei jeder Datei	
-r RECURSIVE	rekursiv die Dateien der Unterordner einschließen	

-p PROJECT

Auf den Parameter -p folgt die Adresse des TETRA-Projekts, mit dem die Dateien des Quellverzeichnisses transformiert werden sollen.

-m MANAGEMENT

Auf den Parameter -m folgt die Adresse des Transformations-Manager-Projekts, in dem die Quell- und Ziel-Dateien spezifiziert sind.
Wird der -m Paramerter angeführt, so werden -s und -t und -r ignoriert.

-s SOURCE

Auf den Parameter -s folgt die Kennzeichnung der Dateien, die transformiert werden sollen.
Diese Kennzeichnung ist im einfachsten Fall die Adresse einer einzelnen Datei, wie "C:\dir\source.txt". Um alle TXT-Dateien eines Verzeichnisses zu transformieren, kann eine Maske wie "C:\dir*.txt" verwendet werden. Die Dateien des aktuellen Verzeichnisses werden transformiert, wenn die Maske keine Verzeichnisangabe enthält. Z.B.:"ab?.*" würde alle Dateien des aktuellen Verzeichnisses als Quelldateien auswählen, sie mit "ab" beginnen, worauf ein einzelnes Zeichen und eine beliebige Erweiterung folgt; etwa "ab1.txt", "ab2.txt" und "ab_.bat"

-t TARGET

Die Angabe eines Ziels ist optional. Fehlt sie, so werden die Quelldateien durch ihrer transformierten Versionen überschrieben. Als Ziel kann ein vollständig spezifizierter Dateiname angegeben werden oder aber ein Dateiname ohne spezifiziertes Verzeichnis. Im letzteren Fall wird die Datei in das Quellverzeichnis geschrieben. Wird umgekehrt zwar ein Zielverzeichnis, aber kein Dateiname angegeben, so werden die Quelldateien unter ihrem alten Namen in das neue Verzeichnis geschrieben.

-b BACKUP

Ein Backup-Verzeichnis ist notwendig, wenn mindestens eine der Quelldateien durch ein Zieldatei überschrieben wird.. Wenn keine Datei überschrieben wird, wird kein Backup angelegt..

Beispiele:

tetra_cl -p exchange.ttp -s feuerbach.txt -b C:\Backup

"feuerbach.txt" wird durch seine transformierte Version überschrieben. Vor der Transformation wird in "C:\Backup" eine Sicherheitskopie von "feuerbach.txt" angelegt..

tetra_cl -p exchange.ttp -s feuerbach.txt -t bachfeuer.txt

Die transformierte Version von "feuerbach.txt" wird in die Datei "bachfeuer.txt" im gleichen aktuelle Verzeichnis geschrieben.

tetra_cl -p exchange.ttp -s *.* -t ..\newdir

Die transformierten Dateien werden in das Unterverzeichnis "..\newdir" geschrieben. Falls das Verzeichnis nicht existiert, wird es automatisch neu angelegt.

Werden durch die Angabe der Quelle mehrere Dateien ausgewählt zugleich aber ein einzelner Dateiname für das Ziel bestimmt, werden alle Ergebnisse in die einzelne Zieldatei geschrieben. Diese muss von allen Quelldateien verschieden sein.

-c CONFIGURE

Auf die Option "-c" folgend können Parameter, die vor dem Start sämtlicher Transformationen benötigt werden, als Bezeichner oder String an das Projekt übergeben werden.. Dort werden sie mit der Funktion ConfigParam gelesen. Ein Config-Parameter wird für alle Dateien gesetzt.

-x EXTRA

Auf die Option "-x" folgend können Parameter, die vor dem Start einer bestimmten Transformation benötigt werden, als Bezeichner oder String an das Projekt übergeben werden.. Dort werden sie mit der Funktion ExtraParam gelesen. Pro Datei kann ein Extra-Parameter gesetzt werden.

-a ASK

Der optionale Parameter "-a" veranlasst, dass vor der Transformierung jeder Datei die Frage auf dem Bildschirm erscheint:

transform: quelle to ziel ? (yes,no,all,cancel)

Die Frage muss durch Drücken des entsprechenden Anfangsbuchstabens beantwortet werden:

y die Transformation wird für diese Datei durchgeführt
n die Transformation wird für diese Datei *nicht* durchgeführt

a die Transformation wird für diese Datei und alle weiteren Dateien durchgeführt
c die Transformation wird für diese Datei und alle weiteren Dateien abgebrochen

-r RECURSIVE

Der optionale Parameter "-r" bestimmt, dass das Quellverzeichnis rekursiv nach Quelldateien durchsucht wird, d.h. dass in allen Unterverzeichnissen des Quellverzeichnisses nach Dateien die zur Maske passen gesucht wird.

9.6 Tastaturkürzel

Hilfe

diese Hilfe aufrufen	F1
Regex Test	F2

Dateioperationen

STRG + O	Datei öffnen
STRG + S	Datei speichern

Navigieren

UMSCHALT + F5	zur Startregel

Compilieren

F5	Startregel Parsen

Ausführen und Debuggen

F6	Nächstes Token
F7	Einzelne Anweisung
F8	Gesamte Routine
F9	Start
F10	Ausführen
F11	Transformations-Manager
STRG + F12	Zurücksetzen
STRG + UMSCHALT + Ziffer	Text-Haltepunkt setzen/entfernen
STRG + Ziffer	zu Text-Haltepunkt springen

Verwaltung

STRG + ALT + I	Einzelnes Skript parsen (isoliert)
STRG + ALT + P	Zusammenhang parsen
STRG + ALT + T	AlleSkripte parsen (total)

STRG + ALT + N	Neu (New)
STRG + ALT + A	Übernehmen (Accept)
STRG + ALT + C	Verwerfen (Cancel)
STRG + ALT + D	Löschen (Delete)
Delete	Löscht in der Liste markiertes Skript
STRG + ALT + M	Kommentar
SHIFT + F3	Nächstes Vorkommen des markierten Textes suchen (Skript übergreifend)

Zwischenablage

Die folgenden Tastaturkürzel gelten, wenn ein Skript in der Liste der Skripte markiert ist. Ist hingegen ein Editor aktiv haben die Kürzel ihre dort übliche Bedeutung als Textoperationen.

STRG+Einfg kopiert das aktuelle Skript in die Zwischenablage. Es kann von dort in das gleiche Projekt unter anderem Namen neu eingefügt werden, oder aber in ein anderes Projekt, welches in einer zweiten Instanz des TextTransformers geladen ist.
Shift+Einfg fügt ein Skript aus der Zwischenablage in die Liste ein. (Die Übernahme muss anschließend bestätigt werden, wobei im Konfliktfall der Name des Skripts zu ändern ist.)

9.6.1 Blockbefehle

Innerhalb der verschiedenen Editfelder (nicht im Kommentarfeld) gelten folgende Tastaturkürzel für Blockbefehle

Tastaturkürzel	Aktion oder Befehl
STRG+K+B	Markiert den Blockanfang
STRG+K+I	Rückt einen Block ein
STRG+K+K	Markiert das Blockende
STRG+K+L	Markiert die aktuelle Zeile als Block
STRG+K+N	Wandelt einen Block in Großbuchstaben um
STRG+K+O	Wandelt einen Block in Kleinbuchstaben um
STRG+K+T	Markiert ein Wort als Block
STRG+K+U	Rückt einen Block aus
STRG+K+Y	Löscht einen markierten Block

TextTransformer

Teil X

10 Skripte

Ein TETRA-Projekt besteht aus einer Anzahl verschiedenartiger Programmtexte:

Tokendefinitionen mittels regulärer Ausdrücke auf der Tokenseite
Regelskripte in einer speziellen TETRA-Skriptsprache auf der Regelseite
C++-Anweisungen, die in die Regelskripte eingebettet sind
Testskripte auf der Testseite

Viele TETRA-Projekte werden mit den unter Punkt zwei und drei genannten Programmtexten auskommen, da literale Token direkt innerhalb der Regelskripte definiert werden können und da Testskripte nur in komplexen Anwendungen angebracht sind.

Die Namen sämtlicher Regelskripte, Elementskripte und Tokendefinitionen eines Projekts müssen voneinander verschieden sein.

10.1 Tokendefinitionen

Im einleitenden Teil wurde bereits kurz auf die Bedeutung der Token für die Syntaxanalyse eingegangen. Auf der Tokenseite des linken Fensterblocks von TETRA können die Token definiert werden.

Die Definition eines Tokens erfolgt innerhalb eines Formulars.
Literale Token können auch direkt innerhalb einer Produktion definiert werden.

Auf den dann folgenden Seiten wird die Syntax der regulären Ausdrücke erklärt.
Es wird zunächst auf eine einfache Untermenge der regulären Ausdrücke eingegangen: die literalen Ausdrücke. Dann werden die regulären Ausdrücke in extenso dargestellt.

10.1.1 Token-Eingabemaske

Die Eingabemaske für eine Tokendefinition umfasst folgende Felder:

Name: eindeutigen Name
Rückgabetyp: C++-Variablentyp
Parameter: C++-Parameterdeklaration
Kommentar: beliebiger Kommentartext
Text: Skripttext des Tokens
Semantische Aktion: Anweisungen

Die Angabe von Namen und Text sind erforderlich. Erst wenn diese Felder gefüllt sind, ist es möglich das Skript in die Verwaltung zu übernehmen oder einen Kommentar zu schreiben.

10.1.1.1 Name

Jedem Tokenskript muss ein Name gegeben werden. Die Namen der Token werden in den Skripten der Produktionen verwendet, um die im Text des Tokenskripts beschriebene Struktur zu bezeichnen.
Der Name kann aus den Buchstaben des Alphabets, Ziffern und Unterstrichen '_' gebildet sein, wobei der Unterstrich nicht an erster Stelle des Namens stehen darf.

Beispiele.: IDENTIFIER, int_value, UB40

Jeder neue Name muss sich in mindestens einem Zeichen von allen anderen Namen der in der Verwaltung enthaltenen Token, Produktionen, Funktionen und Variablen unterscheiden.

10.1.1.2 Rückgabetyp

Den Anweisungen in dem Feld "Semantische Aktion" eines Tokenskripts, die unmittelbar nach Erkennung des Tokens ausgeführt werden, können einen Wert zurückgeben. Für den Typ des Werts gilt das gleiche, was für den Rückgabetyp der Produktionen beschrieben ist.

10.1.1.3 Parameterdeklaration

Den Anweisungen in dem Feld "Semantische Aktion" eines Tokenskripts, die unmittelbar nach Erkennung des Tokens ausgeführt werden, können Parameter übergeben werden. Für diese Parameter gilt das gleiche, was für die Parameter der Produktionen beschrieben ist.

10.1.1.4 Kommentar

Ein Kommentartext zum Token wird in dem gelblich hinterlegten Feld dargestellt.

Dieses Feld wird temporär auch zur Anzeige von Fehlermeldung etc. verwendet. Um den Kommentar zu ändern muss der Kommentarschalter betätigt werden, der eine gesonderte Dialogbox zum edieren des Kommentartextes öffnet.

10.1.1.5 Text

Das Textfeld dient der eigentlichen Definition eines Tokens als regulärer Ausdruck. Am einfachsten ist es, den

gesamten Ausdruck in der ersten Zeile des Feldes

zu platzieren. Dies ist immer möglich und wird in der weitaus überwiegenden Zahl der Fälle

praktiziert. Um die Lesbarkeit der Ausdrücke zu erhöhen können sie aber auch

auf mehrere Zeilen verteilt und kommentiert

werden. Die Teile des Ausdrucks müssen dabei stets am Anfang einer Zeile stehen und die Verbindung der Zeilen muss durch das **Zeilenfortsetzungszeichen " \"** hergestellt werden. Auf jedes der Zeilenfortsetzungszeichen und auf den letzten Teilausdruck kann in der gleichen oder nächsten Zeile ein Zeilenkommentar folgen. **Kommentarzeilen** werden mit einem doppelten Schrägstrich "//" eingeleitet.Die folgenden Schreibweisen sind also äquivalent:

```
\w+::\w+ // class and function name

\w+ \  // class name
::\w+  // function name

\w+ \
// class name
::\w+
// function name
```

Zwischen dem Teilausdruck und dem Fortsetzungszeichen, bzw. einem Zeilenkommentar muss sich mindestens ein Leerzeichen befinden.

10.1.1.6 Semantische Aktion

In dem Feld "Semantische Aktion" eines Tokenskripts können Anweisungen eingegeben werden, die unmittelbar nach Erkennung des Tokens ausgeführt werden. Kommt beispielsweise ein bestimmtes Token, innerhalb des Regelsystems des Parsers häufiger vor, so kann die Behandlung des von dem Token erkannten Textes zentral an dieser Stelle vorgenommen werden. So können z.B. ähnlich einem zweisprachigen Wörterbuch Tokentexten ihre Übersetzungen zugeordnet werden.

```
Text:     "Hello world"
Aktion: out << "Hallo Welt";
```

Für die Syntax der Anweisungen gilt das gleiche, was für die semantischen Aktionen innerhalb der Produktionen beschrieben ist. Werden keine Klammern gesetzt gilt die Projektoption, die für die doppelt geschweifte Klammer "{{...}}" eingestellt ist.

Für die dem Token zugeordneten Aktionen weden im Syntaxbaum keine gesonderten Knoten erzeugt. So wird er übersichtlicher.

Es ist nicht möglich für ein Token eine Übergangsaktion zu definieren, wenn ihm auch eine Aktion in der hier beschrieben Weise zugewiesen ist.

10.1.2 Literale

Jedes spezielle Wort eines Textes, jede Zahl und überhaupt jeder Teil eines Textes kann als ein individuelles literales Token aufgefasst werden. Ein Literal ist schlicht eine bestimmte Folge von Zeichen. Das Wort "TETRA" z.B. ist ein Literal als ein 'T" gefolgt von einem 'E', einem 'T', 'R' und von

einem 'A'.

Entsprechend ihrer Einfachheit und ihrer Bedeutung für die Syntaxanalyse müssen Literale in der TETRA-Verwaltung nicht gesondert auf der Token-Seite definiert und benannt werden, sondern können auch innerhalb der Produktionen direkt definiert werden. Dies ist auf zweifache Wiese möglich:

1. durch Einschluss des Textes in Anführungszeichen, z.B. "TETRA"
2. durch Voranstellen eines Unterstrichs, z.B. _TETRA

Im zweiten Fall wird ein benanntes Token erzeugt, dass automatisch auf der Token-Seite eingefügt wird. Die besonderen Vorteile benannter literaler Token werden gesondert besprochen. Normalerweise reichen die einfachen literalen Token aus. Beispielsweise könnte eine Regel zum Parsen einer Anrede folgendermaßen aussehen:

```
( "Herr" | "Frau" ) name
```

Hier bedeuten "Herr" und "Frau" jeweils die Worte "Herr" und "Frau" selbst, während *name* eine weitere Regel oder einen regulären Ausdruck bezeichnen könnte.

Innerhalb der direkt in den Produktionen definierten Token hat jedes Zeichen literale Bedeutung, d.h. es bedeutet sich selbst. Dies gilt insbesondere auch für die Metazeichen der regulären Ausdrücke. Soll beispielsweise ein Smiley erkannt werden, so kann das direkt folgendermaßen geschehen:

";-)"

Würde das gleiche Smiley als regulärer Ausdruck definiert werden, so wäre seinen Formulierung schwieriger:

";\-\)"

Sowohl dem Bindestrich als auch der Klammer müsste ein Backslash vorangestellt werden, um die Metabedeutung dieser Zeichen auszuschalten.

Einige sonst nicht darstellbare Zeichen, wie z.B. Zeilenumbrüche, können als Escape-Sequenzen auch innerhalb der Definition literaler Token verwendet werden.

10.1.2.1 Benannte Literale

Manchmal ist es von Vorteil, statt einfache literaler Token zu verwenden, benannte literale Token zu definieren. Dies kann entweder direkt auf der Token-Seite erfolgen oder, besonders einfach, die Token werden innerhalb einer Produktion definiert indem dem literalen Text ein Unterstrich vorangestellt wird. In diesem Falle wird bei der Übernahme der Produktion automatisch ein neues Token auf der Token-Seite eingefügt. Wenn z.B. in der Produktion der Ausdruck *_TETRA* vorkommt, so wird daraus auf der Token-Seite folgendes Token:

Name	_TETRA
Definition	TETRA

Da Sonderzeichen in Skript-Namen nicht erlaubt sind, können auf diese Weise keine Token definiert werden, die Sonderzeichen enthalten. Es ist aber möglich, die Definition eines benannten Literals nachträglich entsprechend zu ändern. Z.B.:

Name	_TETRA
Definition	**TETRA**

Beginnt der Name eines Tokens mit einem Unterstrich, so werden alle in der Definition vorkommenden Zeichen literal interpretiert. Die Sterne '*' haben hier also **nicht** die Bedeutung von Wiederholungszeichen.

Die Vorteile benannter literaler Token sind, dass sie

1. einen eindeutigen Namen haben,
2. ihnen eine semantische Aktion zugeordnet werden kann und
3. diese Namen auch im erzeugten C++-Code verfügbar sind.

zu 1.
Dadurch, dass benannte literale Token einen eindeutigen Namen haben, können Schreibfehler vermieden werden, die unterlaufen könnten, wenn das gleiche Literal an verschiedenen Stellen der Grammatik verwendet wird.

zu 2.
Wenn stets die gleiche semantische Aktion beim Antreffen eines bestimmten literalen Tokens im Text ausgeführt werden soll, auch wenn es mehrmals an verschiedenen Stellen der Grammatik vorkommt, dann ist es angebracht, die Möglichkeit auszunutzen, ein Token direkt mit einer Aktion zu verknüpfen. Es ist z.B. oft sinnvoll, bei der Erzeugung von Parse-Bäumen die Texte der literalen Token auf stets gleiche Weise als "Blätter" in den Baum einzufügen.

zu 3.
Bei der Generierung von C++-Code aus einem Projekt wird für jedes Literal eine Zeichenkette erzeugt, der auch in Code des Programms verwendet werden kann, das den Parser benutzt. Z.B.

```
const char _protected[] = "protected";
```

10.1.3 Reguläre Ausdrücke

Der TextTransformer verwendet die regular expression library von Dr John Maddock. DieSyntax der regulären Ausdrücke hier ist daher - mit kleinen Einschränkungen - die gleiche, wie dort.

Im Unterschied zu den Literalen bedeuten in komplexeren regulären Ausdrücken nicht alle Zeichen sich selbst. In ihnen werden eine Reihe von Metazeichen verwendet, die der Definition von Zeichenmengen, der Gruppierung von Zeichen und der Definition von Abfolgen dienen.

Folgende Elemente werden zur Definition regulärer Ausdrücke verwendet:

10.1.3.1 Einzelzeichen

Alle Zeichen, die keine spezielle Meta-Bedeutung haben passen auf sich selbst. So wird z.B. mit dem Zeichen 'A' im regulären Ausdruck der Buchstabe 'A' im Quelltext erkannt und wenn der reguläre Ausdruck die Zeichenfolge "Hallo" enthält, wird damit das Wort "Hallo" erkannt.

10.1.3.2 Metazeichen

Metazeichen sind:

'.', '|', '*', '?', '+', '(', ')', '{', '}', '[', ']', '^', '$' und '\'

Wird eines dieser Metazeichen in seiner literalen Bedeutung benötigt, so wird ihm innerhalb der regulären Ausdrucks ein Backslash '\' vorangestellt. Der Backslash hebt die Metabedeutung eines Zeichens auf.

Zitieren

Mit der Escape-Sequenz \Q beginnt ein "Zitat": alle nachfolgenden Zeichen werden als Literale behandelt, entweder bis zum Ende des regulären Ausdrucks oder bis \E gefunden wird. Zum Beispiel würde der Ausdruck: \Q*+\Ea+ auf die beiden folgenden Zeilen passen:

```
\*+a
\*+aaa
```

10.1.3.3 Spezielle Zeichen

Auf dem Bildschirm nicht darstellbares Zeichen können durch Escape-Sequenzen repräsentiert werden.

Die Regeln für die Verwendung von hexadezimalen Zeichen sind gegenüber der Verwendung in allgemeinen Zeichenketten etwas modifiziert. Nur zwei hexadezimale Ziffern werden als ein Zeichen interpretiert, wenn sie nicht geklammert sind.

\xdd passt auf ein einzelnes Zeichen 0xdd
\x{dddd} passt auf ein einzelnes Zeichen 0xdddd

\N{Name} passt auf ein einzelnes Zeichen, das den symbolischen Namen *name* hat. Zum Beispiel \N{newline} passt auf das einzelne Zeichen \n.

10.1.3.4 Zeichnmengen

Eine Zeichenmenge ist eine Menge von Zeichen, die jeweils ein Zeichen des Eingabetexts erkennen kann, das in der Menge enthalten ist.

Zeichenmengen werden definiert, indem die Zeichen, die sie enthalten soll zwischen eckige Klammern "**[...]**" gesetzt werden. Die Zeichen, die die Menge enthält können dabei entweder einzeln aufgelistet werden, oder es können Bereiche von Zeichen angegeben oder bereits vordefinierte Zeichenmengen eingesetzt werden.

Die Zeichen ".|*?+(){}$", die sonst innerhalb der regulären Ausdrücke eine Metabedeutung haben, haben innerhalb der Definition von Zeichenmengen ihre literale Bedeutung, d.h. ihnen muss hier kein Backslash vorangestellt werden.

Zeichenmengendefinitionen, die mit **"^"** beginnen, enthalten das Komplement zu den folgenden Elementen.

Beispiele:

Zeichenlisten:

"[abc]" passt auf jedes der Zeichen 'a', 'b' oder 'c'.

"[^abc]" passt auf jedes andere Zeichen als 'a', 'b' oder 'c'.

Zeichenbereiche:

"[a-z]" passt auf jedes Zeichen von 'a' bis 'z'.

"[0-9]" passt auf jede Ziffer.

"[^A-Z]" passt auf jedes Zeichen, dass nicht im Bereich von 'A' bis 'Z' liegt.

Kombinationen:

Alles obige zusammen mit vordefinierten Zeichenmengen und symbolische Namen können zur Deklaration einer Zeichenmenge kombiniert werden; zum Beispiel:

[[:digit:]a-c[.NUL.]].

Um das Zeichen '-' selbst in eine Zeichenmenge einzuschließen, stellt man ihm den Backslash voran.

Um eines der Zeichen '[' oder ']' oder '^' in eine Zeichenmenge einzuschließen, ist ihnen der Backslash voran zustellen.

Anm.: Es ist auch möglich, '-' oder '[' oder ']' oder '^' ohne Backslash als Bereichsgrenze zu verwenden.
Das Zeichen '-' kann auch ohne Backslash direkt hinter die öffnende Klammer '[' oder '[^' gesetzt werden..

10.1.3.5 vordefinierte Zeichenmengen

Es gibt einige vordefinierte Zeichenklassen. Sie können innerhalb einer Zeichenmengendefinition verwendet werden, indem ihr Name in speziellen Klammern gesetzt wird: "[:Zeichenklassenname:]". Die Menge der Ziffern z.B. heißt "dgit" und eine Zeichenmenge: [[:digit:],] besteht aus allen Ziffern und dem Kommazeichen.

Folgende Zeichenklassen sind definiert:

alnum	die alphanumerischen Zeichen, *alpha* und *digit* (*)
alpha	die Zeichen des Alphabets a-z und A-Z, Umlaute etc. (*)
blank	das Leerzeichen, das geschützte Leerzeichen (dezimal 160) und der Tabulator
cntrl	die Steuerzeichen
digit	die Ziffern 0-9
graph	alle grafischen Zeichen, alle anderen außer *cntrl*
lower	die Kleinbuchstaben des Alphabets a-z (*)
print	alle druckbaren Zeichen; *graph* und *blank*
punct	die Interpunktionszeichen
space	die Leerzeichen (Leerzeichen, Tabulator, Wagenrücklauf, Zeilenvorschub, vertikaler Tabulator oder ein Seitenvorschub)
upper	die Großbuchstaben des Alphabets A-Z (*)
xdigit	die Hexadezimalzeichen, 0-9, a-f und A-F
word	die alphanumerischen Zeichen und der Unterstrich (*)

(*) entsprechend den lokalen Einstellungen auf Ihrem Computer können noch weitere Zeichen erkannt werden. Probieren Sie es mit dem Dialog zur Berechnung von Zeichenmengen aus!

Für einige dieser Zeichenklassen und ihre Komplementklassen gibt es wiederum eine abgekürzte Schreibweise

\w	[:word:]
\W	^[:word:]
\s	[:space:]
\S	^[:space:]
\d	[:digit:]
\D	^[:digit:]
\l	[:lower:]
\L	^[:lower:]
\u	[:upper:]
\U	^[:upper:]

10.1.3.6 Länderspezifische Merkmale

Für reguläre Ausdrücke gibt es einige länder- bzw. sprachspezifische Definitionen. Das betrifft ihre Verwendung im TextTransformer nur am Rande, weil er zur Zeit nur die deutsche Lokalisierung - wenn diese auf ihrem Computer installiert ist - oder sonst die englische unterstützt und nur den ANSI-Zzeichensatz verwendet. In speziellen Fällen kann aber doch von diesen Merkmalen Gebrauch gemacht werden und für eine erweiterte Nutzung des generierten Codes kann eine Kenntnis dieser Eigenschaften ebenfalls hilfreich sein.

Kollationierende Elemente
Äquivalenz-Klassen
Namen kollationierender Elemente

10.1.3.6.1 Kollationierende Elemente

Ein Ausdruck der Form [[.col.]] passt auf das kollationierende Element col. Ein kollationierendes Element (kollationieren = zusammentragen, vergleichen) ist jedes einzelne Zeichen oder jede Zeichenfolge, die zu einer Einheit kollationiert. Kollationierende Elemente können auch als Begrenzung eines Bereichs verwendet werden, z.B.: [[.ae.]-c] passt auf die Zeichenfolge "ae", und auf jedes einzelne Zeichen im Bereich "ae"-c, unter der Annahme, dass "ae" in der Lokalisierungseinstellung als einzelnes kollationierendes Element gilt.
Kollationierende Elemente können anstelle von Backslashes bei der definition von Zeichenklassen verwendet werden. z.B. würde [[.^.]abc] jedes der Zeichen 'abc^' erkennen.
Ein kollationierendes Element kann auch durch seinen symbolischen Namen spezifiziert werden, z.B.:

```
[[.NUL.]]
```

passt auf ein NUL-Zeichen.

10.1.3.6.2 Äquivalenz-Klassen

Ein Ausdruck der Form [[=col=]], passt auf jedes Zeichen oder kollationierende Element, dessen primärer Sortierungsschlüssel der gleiche ist, wie für das kollationierende Element *col*. Der Name *col* kann auch ein symbolischer Name sein. Ein primärer Sortierungsschlüssel ist einer, der Groß-/Kleinschreibung, Akzentuierungen und lokalitätsspezifische Einstellungen ignoriert; so passt [[=a=]] zum Beispiel auf jedes der Zeichen: a, à, á, â, ã, ä, å, A, À, Á, Â, Ã, Ä und Å.
Leider beruht die Implementierung der Äquivalenz-Klassen auf der Unterstützung der Kollationierung und Lokalisierung durch die jeweilige Plattform. Man kann sich daher nicht darauf verlassen, dass die Äquivalenz-Klassen über verschiedene Plattformen oder auch nur verschiedene Lokaliserungen einer Plattform portabel sind.

10.1.3.6.3 Namen kollationierender Elemente

Digraphen (Verbindung von zwei Buchstaben zu einem Laut)

Folgendes wird als gültige Digraphen behandelt, wenn es als kollationierender Namen verwendet wird:

"ae", "Ae", "AE", "ch", "Ch", "CH", "ll", "Ll", "LL", "ss", "Ss", "SS", "nj", "Nj", "NJ", "dz", "Dz", "DZ", "lj", "Lj", "LJ".

POSIX Symbolische Namen

Die folgenden symbolischen Namen werden als gültige kollationierende Elemente erkannt, zusätzlich zu jedem Einzelzeichen:

Name	Character
NUL	\x00
SOH	\x01
STX	\x02
ETX	\x03
EOT	\x04
ENQ	\x05
ACK	\x06
alert	\x07
backspace	\x08
tab	\t
newline	\n
vertical-tab	\v
form-feed	\f
carriage-return	\r
SO	\xE
SI	\xF
DLE	\x10

DC1	\x11
DC2	\x12
DC3	\x13
DC4	\x14
NAK	\x15
SYN	\x16
ETB	\x17
CAN	\x18
EM	\x19
SUB	\x1A
ESC	\x1B
IS4	\x1C
IS3	\x1D
IS2	\x1E
IS1	\x1F
space	\x20
exclamation-mark	!
quotation-mark	"
number-sign	#
dollar-sign	$
percent-sign	%
ampersand	&
apostrophe	'
left-parenthesis	(
right-parenthesis	)
asterisk	*
plus-sign	+
comma	,
hyphen	-
period	.
slash	/
zero	0
one	1
two	2
three	3
four	4
five	5
six	6
seven	7
eight	8
nine	9
colon	:
semicolon	;
less-than-sign	<

equals-sign	=
greater-than-sign	>
question-mark	?
commercial-at	@
left-square-bracket	[
backslash	\
right-square-bracket	]
circumflex	~
underscore	_
grave-accent	`
left-curly-bracket	{
vertical-line	|
right-curly-bracket	}
tilde	~
DEL	\x7F

10.1.3.7 Punkt

Durch den Punkt "." wird ein beliebiges einzelnes Zeichen erkannt. Man kann ihn lesen als eine Zeichenmenge, die sämtliche Zeichen umfasst.

10.1.3.8 Anker

Achtung: Die Wirksamkeit der Anker hängt von der Einstellung der auszulassenden Zeichen ab, und davon, ob der sie verwendende Ausdruck Ziel eines SKIP-Knotens ist oder nicht. (s. Anmerkung unten)

Anker sind Elemente regulärer Ausdrücke, die markante Positionen im Text erkennen. Die Länge des von einem Anker erkannten Textes ist null.

Zeilenanker

'^' passt zum Beginn einer Zeile, wenn es als erstes Zeichen eines Ausdrucks oder eines Unterausdrucks verwendet wird.

'$' passt zum Zeilenende, wenn es als letztes Zeichen eines Ausdrucks oder eines Unterausdrucks verwendet wird.

Wortanker

"\<" passt zum Wortanfang.

"\>" passt zum Wortende.

"\b" passt zum Wortanfang oder -ende.

"\B" passt zu einer Position innerhalb eines Wortes.

Pufferanker

Der Parsprozeß in TETRA erfolgt linear mit dem Text. In jedem Stadium dieses Prozesses befindet sich dieser an einer bestimmten Position im Text, die den bereits abgearbeiteten Textteil von dem noch nicht analysierten Teil trennt. Auf den letzteren Teil lassen sich die Pufferanker anwenden.

"\`" oder "\A" passen zum Beginn des Puffers.

"\'" oder "\z" passen zum Ende des Puffers.

"\Z" passt zum Ende des Puffers oder zu einem oder mehreren Zeilenumbruchszeichen unmittelbar vor dem Pufferende.

Anmerkung:

Der Pufferanker "\A" wird in TETRA automatisch allen regulären Ausdrücken vorangestellt. Das jeweils nächste Token - der Suchpuffer - soll dann an der aktuellen Textposition im Anschluss an die auszulassenden Zeichen beginnen. Damit wird die Verwendung des Ankers "^" hinfällig. Für den regulären Ausdruck ist der Beginn des Suchpuffers zugleich ein Zeilenanfang, auch wenn der Suchpuffer in der Mitte einer Zeile des Eingabetextes beginnt.
Dennoch kann die Verwendung von "^" Sinn machen, wenn der Anker eines der Sprungziele eines SKIP-Knotens markiert.

Sei ein Token definiert als:

LINE_START_WORD = ^\w+

und enthalte die Grammatik:

SKIP LINE_START_WORD

so wird die erste Zeile des folgenden Textes übersprungen:

" Wort1 steht nicht am Zeilenanfang
Wort2 steht am Zeilenanfang"

und "Wort2" wird als LINE_START_WORD erkannt.

10.1.3.9 Verkettung

Bisher wurden nur Elemente von regulären Ausdrücken erläutert, die jeweils auf ein einzelnes Zeichen passen. Aus diesen und den nachfolgend erklärten Elementen lassen sich komplexe reguläre Ausdrücke bilden, die in der Lage sind Zeichenketten, also z.B. Worte, Zahlen etc zu erkennen. Die einfachste Art hierfür ist die Aneinanderreihung der regulären Ausdruckselemente derart, dass jedes Element ein Zeichen der zu erkennenden Zeichenkette beschreibt.

Der reguläre Ausdruck "TETRA" besteht aus der Aneinanderreihung der regulären Ausdruckselementen "T", "E", "T", "R" und "A" und erkennt eine Zeichenfolge aus den Buchstaben

"T", "E", "T", "R" und "A", also das Wort "TETRA". Das gleiche Wort würde auch von dem regulären Ausdruck "[A-Z]E[^0-9]RA" erkannt. Dieser Ausdruck ist allgemeiner als der erste und würde auf viele andere Texte passen, z.B. "AE&RA" oder "HEDRA". Das Prinzip ist aber das gleiche: jedes der aufeinander folgenden Elemente des regulären Ausdrucks erkennt das in der Folge entsprechende Zeichen des Textes.

10.1.3.10 Gruppierung

Die Klammern '(' und ')' dienen zweierlei Zwecken:

1. sie gruppieren Elemente des regulärer Ausdrücke zu Unterausdrücken.

Die Gruppierung zu Unterausdrücken hat den Sinn, dass auf einen solchen Unterausdruck insgesamt Operatoren wie z.B. der Wiederholungsoperator "*" (s.u.) angewandt werden können.

2. sie markieren Teile des erkannten Textes.

Wird eine Übereinstimmung eines komplexen regulären Ausdrucks mit einem Text gefunden, so kann über xState sowohl ermittelt werden, was durch den gesamten Ausdruck erkannt wurde, als auch, was durch jeden seiner Unterausdrücke erkannt wurde.
Unterausdrücke werden beginnend mit 1 von links nach rechts im gesamten Ausdruck gezählt. Der Ausdruck mit dem Index 0 ist der gesamte Ausdruck selbst. Die entsprechenden Abschnitte erhält man durch

```
str s = xState.str(index);
```

Wenn ein Unterausdruck an der Übereinstimmung des gesamten Ausdrucks keinen Anteil hat- z.B. wenn er Teil einer nicht gewählten Alternative ist - gibt xState.str(index) einen Leersting zurück.

10.1.3.11 Alternativen

Ein regulärer Ausdruck (oder Unterausdruck) kann Alternativen enthalten. D.h. der gesamte Ausdruck steht dann in Übereinstimmung mit dem Text, wenn eine seiner Alternativen auf den Text passt. Alternativen werden durch das Pipe-Zeichen "|" voneinander getrennt.

Beispiele:

"a(b|c)" passt auf "ab" oder "ac".

"abc|def" passt auf "abc" oder "def", aber nicht auf abdef.

Am letzten Beispiel ist ersichtlich, dass jede Alternative den größtmöglichen Unterausdruck umfasst (im Gegensatz zu den Wiederholungen s.u.): "abc|def" ist nicht etwa zu lesen als "ab" gefolgt von "c" oder "def", sondern als "abc" oder "def".

10.1.3.12 Wiederholung

Jedes Atom (ein einzelnes Zeichen, ein markierter Unterausdruck, oder eine Zeichenmenge) kann mit den *, +, ?, und {} Operatoren wiederholt werden.

*****

Der * Operator stimmt mit einer null- oder mehrmaligen Wiederholung des vorausgegangenen Atoms überein. Z.B. wird der Ausdruck a*b jeden der folgenden Texte erkennen:

```
b
ab
aaaaaaaab
```

+

Der + Operator stimmt mit einer ein- oder mehrmaligen Wiederholung des vorausgegangenen Atoms überein. Z.B. wird der Ausdruck a+b jeden der folgenden Texte erkennen:

```
ab
aaaaaaaab
```

Aber nicht:

```
b
```

Der +-Operator darf nicht auf löschbare Strukturen angewendet werden.

?

Der ? Operator stimmt mit einer null- oder einmaligen Wiederholung des vorausgegangenen Atoms überein. Z.B. wird der Ausdruck ca?b jeden der folgenden Texte erkennen:

```
cb
cab
```

Aber nicht:

```
caab
```

{}

Ein Atom kann auch begrenzt wiederholt werden:

a{n} passt aus eine genau n-malige Wiederholung von 'a'.

a{n,} passt aus eine n- oder mehr-malige Wiederholung von 'a'.

a{n, m} passt auf eine n- bis m-malige (inklusiv) Wiederholung von 'a'

Zum Beispiel:

```
^a{2,3}$
```

passt auf beide:

```
aa
aaa
```

aber auf keines von:

```
a
aaaa
```

Die Benutzung eines Wiederholungsoperators ist ein Fehler, wenn sich das vorausgehende Konstrukt nicht wiederholen lässt. Zm Beispiel:

```
a(*)
```

Alle Wiederholungsausdrücke ("+", "*" und "?") beziehen sich auf den kürzesten vorherigen Unterausdruck (im Gegensatz zu den Alternativen s.o.); beispielsweise ein einzelnes Zeichen, eine Zeichenmenge oder einen Unterausdruck, der durch Klammern "(...)" zusammengefasst ist. Der erste Beispielsausdruck "ba*" passt daher nicht auf "baba".

10.1.3.13 Makros

Die Namen bereits definierter Token können als Makro zur vereinfachten Definition weiterer Token verwendet werden. Hierzu wird der Name in die geschweiften Klammern '{' und '}' eingeschlossen und dieser Ausdruck an der entsprechenen Stelle der neuen Definition eingesetzt. Diesen Ausdruck ersetzt der TextTransformer dann beim Parsen der neuen Definition durch den Definitionstext des ersten Tokens.

Beispiel:

```
SPACES = [ \t]*

DECLARATOR =
                    (((\w+::)*\w+)::)?(\w+) \ //Scope(s) und Name
                    {SPACES} \              // optionale Leerzeichen
                    \([^)]*\)               // Parameter
```

Vom TextTransformer werden die Zeilen der Tokendefinition intern zu einer einzigen Zeile zusammengezogen. Das ergibt dann:

```
(\w+::)?(\w+)::(\w+)[ \t]*\([^)]*\)
```

Die Möglichkeit über den Tokennamen auf andere Tokendefinitionen zu rekurrieren, erlaubt es z.B. eine spezielle Zeichenklasse zu definieren, die dann in verschiedenen Tokendefinitionen verwendet wird.

10.1.3.14 boost regular expression library

Der TextTransformer verwendet die regular expression library von Dr John Maddock (http://www.boost.org), die einen Teil der gesamten boost-Bibliothek ausmacht. Die Syntax der regulären Ausdrücke hier, ist daher - mit kleinen Einschränkungen - die gleiche, die dort als POSIX-Extended Regular Expression Syntax beschrieben ist.
Die boost regular expressions können durch Flags modifiziert werden. Im TextTransformer wurden mit zwei Ausnahmen das Standard-Flag (= extended) beibehalten.

```
const tt_syntax_option_type _boost_regex_normal =
      boost::regex_constants::extended
      & ~boost::regex_constants::no_escape_in_lists
      & ~boost::regex_constants::collate;
```

Das Flag "no_escape_in_lists" wurde negiert. D.h. bei der Definition von Zeichenmengen muss dem Backslash-Zeichen selbst ein Backslash vorangestellt werden.

Das Flag "collate" wurde negiert, damit Zeichenbereiche in der Ordnung definiert werden können, wie die Zeichen in der ANSI-Tabelle aufgelistet sind.

Durch das *flag* "extended" wird spezifiziert, dass die regulären Ausdrücke auf gleiche Weise arbeiten wie POSIX reguläre Ausdrücke. D.h der Erkennungsalgorithmus arbeitet so, dass der erste mögliche String im Text erkannt wird, und dass im Falle verschiedener Übereinstimmungen an der gleichen Position die Möglichkeit mit der längsten Übereinstimmung ausgewählt wird. Dieser Algorithmus ist für den TextTransformer wesentlich. Um ihn beibehalten zu können, muss leider auf die bisher mögliche Verwendung von nicht gierigen Wiederholungen und einer Kontextvorausschau verzichtet werden. Rückrferenzen sind nicht explizit verboten, aber sie werden normaler Weise nicht korrekt funktionieren, da die Nummerierung von Unterausdrücken in SKIP-ausdrücken verschoben ist. Rückrferenzen sollten nicht verwendet werden.

10.1.4 Vordefinierte Token

Wenn sie sich mit der Maus in der Tokenliste amlinken Rand des Tokenfensters befinden, erhalten sie nach Drücken der rechten Maustaste eine Anzahl vordefinierter Token angezeigt. Wenn sie eines dieser Token auswählen, wird ein neues Skript mit gefülltem Namens- und Text-Feld geöffnet. Sie können das Skript noch modifizieren oder direkt übernehmen.

Die vordefinierten Token sind in folgende Kategorien aufgeteilt:

Bezeichner
Worte
Zahlen
Anführungen

Datumsangaben
Kommentare
Auszulassendes
Zeilenumbruch
Binäre Null
Adressen
Daten-Feld (Zeichenmengen-Berechner)

10.1.4.1 Bezeichner

Die folgenden vordefinierten Token können über ein Popup-Menü in ein Projekt eingefügt werden.

ID: `[a-zA-Z_]\w*`

Bezeichner, der mit einem Zeichen des Alphabets oder dem Unterstrich beginnt, dem in beliebiger Anzahl alphanumerische Zeichen oder Unterstriche (= \w) folgen können. Wichtig ist, dass ein Bezeichner nicht mit einer Ziffer beginnen darf, da er dann auch Zahlen abdecken würde, für die sich meist ein gesondertes Token empfiehlt.

URI_WS_DELIM
URI_QUOTE_DELIM
URI_ANGLE_DELIM

Diese regulären Ausdrücke stehen für einheitliche Bezeichner für Ressourcen, wie z.B. die Adressen von Webseiten, die gemäß der Norm RFC 3986

http://www.apps.ietf.org/rfc/rfc3986.html

definiert sind. Auf Englisch heißen diese Bezeichner "Uniform Resource Identifier" (URI).
Die drei Ausdrücke sind Varianten eines auf der genannten Seite gegebenen Ausdrucks. Im Unterschied zu letzterem erkennen diese Ausdrücke keinen leeren Text. Eine URI muss mit dem dort beschriebenen "scheme"-Ausdruck beginnen, d.h. mit Zeichen, auf die ein Doppelpunkt folgt. Außerdem muss eine URI vom umgebenen Text unterschieden werden können. In der Art dieser Abtrennung unterscheiden sich die hier gegebenen drei Ausdrücke.

URI_WS_DELIM : (([^:/?#]+):)(//([^/?#\\s]*))?([^?#\\s]*)(\\?([^#\\s]*))?(#([\\s]*))?

Diese URI wird durch Leerzeichen vom umgebenen Text unterschieden. D.h. die URI selbst darf keine Leerzeichen enthalten. *URI_WS_DELIM* erkennt so z.B. den folgenden Text:

```
http://www.ics.uci.edu/pub/ietf/uri/#Related
```

URI_QUOTE_DELIM : "(([^:/?#]+):)(//([^/?#"]*))?([^?#"]*)(\\?([^#"]*))?(#([^"]*))?"

Diese URI wird durch Anführungszeichen vom umgebenen Text unterschieden. D.h. die URI selbst könnte so geschrieben werden, dass sie Leerzeichen enthielte. *URI_QUOTE_DELIM* erkennt so z.B. den folgenden Text:

```
"http://www.ics.uci.edu/pub/ietf/uri/#Related"
```

URI_ANGLE_DELIM : <(([^:/?#]+):)(//([^/?#>]*))?([^?#>]*)(\\?([^#>]*))?(#([^>]*))?>

Diese URI wird duch Leerzeichen vom umgebenen Text unterschieden. D.h. die URI selbst könnte so geschrieben werden, dass sie Leerzeichen enthielte. *URI_ANGLE_DELIM* erkennt so z.B. den folgenden Text:

```
<http://www.ics.uci.edu/pub/ietf/uri/#Related>
```

Anmerkung:

Durch die Unterausdrücke werden im obigen Beispiel die folgenden Abschnitte erkannt:

$1 = http:
$2 = http
$3 = //www.ics.uci.edu
$4 = www.ics.uci.edu
$5 = /pub/ietf/uri/
$6 = <undefined>
$7 = <undefined>
$8 = #Related
$9 = Related

wobei <undefined> anzeigt, dass diese Komponente nicht enthalten ist. Die in RFC 3986 beschriebenen Komponenten sinn dann wie folgt zuzuordnen:

scheme = $2
authority = $4
path = $5
query = $7
fragment = $9

10.1.4.2 Worte

Die folgenden vordefinierten Token können über ein Popup-Menü in ein Projekt eingefügt werden.

Die Mengen möglichen Erkennungen der Worte verschiedener Sprachen überlappen sich.Es sollte nur eine Sprache in einem Projekt verwendet werden, es sei denn, man weiß genau was man tut.

WORD_EN: `[a-zA-Z]+`

WORD_GE: `[a-zA-ZäöüÄÖÜ][a-zA-ZäöüÄÖÜß-]*`

WORD_FR: `[a-zA-ZáàâéèêíìîóòôúùûÀÁÂÈÉÊÌÍÎÒÓÔÙÚÛÇçŒœ]+`

WORD_SP: `[a-zA-ZáéíóúÁÉÍÓÚñ]+`

Englisches, deutsches, französisches oder spanisches Wort. Der reguläre Ausdrück beteht jeweils in der ein- oder mehrmaligen Wiederholung von Buchstaben des jeweiligen Alphabets. Die Buchstaben [a-z) und [A-Z] (= [[:alpha:]] sind dabei in allen Alphabeten enthalten. (Deutsche Worte können nicht mit 'ß' anfangen und können auch Bindestriche enthalten.)

NAME_INT: `[[:alpha:]][-[:alpha:]]*`

Internationaler Name, der auch Bindestriche enthalten kann

10.1.4.3 Zahlen

Die folgenden vordefinierten Token können über ein Popup-Menü in ein Projekt eingefügt werden.

INT: `\d+`

Natürliche Zahlen (Integer-Werte) bestehen schlicht aus einer oder mehreren Ziffern. Die Menge der Ziffern ist vordifiniert als '\d'.

REAL: `(\d+\.\d*|\.\d+)([eE][\+\-]*\d+)?`

Eine reele Zahl, beginnt entweder mit einem Punkt, oder mit einem Punkt, dem eine Integer-Zahl vorausgeht. Auf den Punkt können beliebig viele Ziffern folgen und optional ein Exponenten-Ausdruck. Ein Punkt ohne angrenzende Ziffer ist keine reele Zahl.

Beispiele: .1; 1.;1.1; 3.14, 6.626E-34

HEX_PAS: `\$[[:xdigit:]]{1,8}`

Eine Hexadezimal-Zahl in der Programmiersprache Pascal: ein Dollarzeichen gefolgt von bis zu acht Ziffern oder Buchstaben aus dem Bereich 'A' bis 'F' oder 'a' bis 'f'.

HEX_CPP: `\$[[:xdigit:]]+`

Eine Hexadezimal-Zahl in der Programmiersprache C++: "0x" gefolgt von mehreren Ziffern oder Buchstaben aus dem Bereich 'A' bis 'F' oder 'a' bis 'f'.

10.1.4.4 Anführungen

Die folgenden vordefinierten Token können über ein Popup-Menü in ein Projekt eingefügt werden.

STRING: `"([^"\\\r\n]*(\\.[^"\\\r\n]*)*)"`

String, der mit Anführungszeichen (\") beginnt und endet. Ein String kann weitere Anführungszeichen enthalten, wenn ihnen ein Backslash vorausgeht. Z.B.:

"Dieser Ausdruck wurde gemäß \"Friedl\" optimiert"

Der erste Unterausdruck enthält die Zeichen zwischen den äußeren Anführungszeichen, d.h. durch xState.str(1) erhält man direkt den angeführten Text selbst.

Der hier definierte String kann sich nicht über mehere Zeilen erstrecken.

Anmerkung: Dieser Ausdruck ist gemäß dem Friedl-Schema formuliert.

CHAR_CPP: `'(\\?.)'`

In einfache Anführungszeichen eingeschlossene Zeichen werden in C++ verwendet. Wenn ihnen ein Backslash vorausgeht, erhalten sie eine besondere Bedeutung, wie z.B. das Zeilenumbruchszeichen: '\n'.

10.1.4.5 Datumsangaben

Die folgenden vordefinierten Token können über ein Popup-Menü in ein Projekt eingefügt werden.

Die Mengen möglichen Erkennungen der folgenden Datumsformate überlappen sich. Es sollte nur eine Format in einem Projekt verwendet werden, es sei denn, man weiß genau was man tut. Die Ausdrücke sind allgemein gehalten. So werden neben 4-stelligen auch 2.stellige Jahresangaben erkannt und es sind verschiedene Trenner zugelassen. Für eine konkrete Anwendung sollten die Ausdrücke soweit möglich spezifiziert werden.

DD_MM_YYYY:

```
(0[1-9]|[12][0-9]?|3[01]?|[4-9])[-./] \// day
(0[1-9]|1[0-2]?|[2-9])[-./] \//month
(\d\d|\d{4,4}) //year
```

Dieser Ausdruck erkennt Datumsangaben wie: 03.02.56; 3.2.1856; ungültige Datumsangaben, wie der 31.2.9999, sind nicht ausgeschlossen.

Anmerkung: Jeder der Klammern des Ausdrucks umfasst gerade einen Datumsteil. So können Tag, Monat und Jahr leicht erhalten werden durch:

```
str sTag = xState.str(1);
str sMonat = xState.str(2);
str sJahr = xState.str(3);
```

Der reguläre Ausdruck ist zur Optimierung nach dem LL(1)-Prinzip aus einander ausschließenden Alternativen aufgebaut. Z.B. die möglichen 31 Tage eines Monats beginnen entweder mit einer vorangestellten 0 oder mit einer der anderen Ziffern auf die jeweils spezifische weitere Ziffern folgen

können:

Alternative	Beginn	Wertebereich
0 **[** 1-9 **]**	**0**	01, 02, 03, 04, 05, 06, 07, 08, 09
| **[** 12 **]** **[** 0-9 **]?**	**1**	1 oder 10 - 19
	2	2 oder 20 - 29
| 3 **[** 01 **]?**	**3**	3 oder 30 oder 31
| **[** 4-9 **]**	**4**	4
	5	5
	6	6
	7	7
	8	8
	9	9

YYYY_MM_DD:

```
(\d\d|\d{4,4})[-./] \// year
(0[1-9]|1[0-2]?|[2-9])[-./] \// month
(0[1-9]|[12][0-9]?|3[01]?|[4-9]) //day
```

Dieser Ausdruck ist eine Umstellung des obigen. Er erkennt Datumsangaben wie: 56.03.02; 1856.3.2; ungültige Datumsangaben, wie der 9999.31.2, sind nicht ausgeschlossen.

10.1.4.6 Kommentare

Die folgenden vordefinierten Token können über ein Popup-Menü in ein Projekt eingefügt werden.

Sie beschreiben Kommentare verschiedener Programmiersprachen. Häufig werden sie als Bestandteil der auszulassenden Zeichen verwendet. In diesem Fall brauchen und können sie nicht mehr in den Produktionen eingesetzt zu werden.

LC: `//[^\n]*(\n|\z)`

Zeilenkommentare, die mit \"//\" beginnen und sich bis zum Ende der Zeile oder der Datei erstrecken.

BC_CPP: `/\*[^*]*\*+([^/*][^*]*\*+)*/`

Block-Kommentar (C++-Stil), der mit "/*" beginnt und mit "*/" endet.
Innerhalb des Kommentars sind Folgen von '*' zulässig, also z.B.: /********/. Die Konstruktion dieses Ausdrucks ist schwer zu verstehen. Sie wird abgeleitet in dem Buch J.E.F. Friedl: Reguläre Ausdrücke. ("[^/*]" nimmt hier die Stelle von *speziell* ein.)

BC_PAS: `\(\*[^*]*\*+([^)*][^*]*\*+)*\)`

Block-Kommentar (Pascal-Stil), der mit "(*" beginnt und mit "*)" endet.
Innerhalb des Kommentars sind Folgen von '*' zulässig, also z.B.: (********)

BC_HTML: `<!--([^-]|-+[^->]|->)*-+->`

HTML/XML Block-Kommentar, beginnend mit "<!--" und schließend mit "-->".

10.1.4.7 Auszulassendes

Die folgenden vordefinierten Token können über ein Popup-Menü in ein Projekt eingefügt werden. Anschließend können sie in den Projekteinstellungen für die auszulassenden Zeichen eingesetzt werden.

IGNORE_CPP:

```
(\s \// spaces
|/)      \// begin of a comment
/[^\r\n]*\$   \// line comment
|\*[^*]*\*+([^/*][^*]*\*+)*/ \// block comment
)   \
)+
```

Zeilenkommentare, Blockkommentare, Leerzeichen und Zeilenumbrüche werden in C++-Code ignoriert.

IGNORABLE_PAS:

```
( \
\s \// spaces
|\{[^}]*\} \// {...}-comment
|\(\*[^*]*\*+([^)*][^*]*\*+)*\) \// (*...*)-comment
)+
```

{...}-Kommentare, (*...*)-Kommentare, Leerzeichen und Zeilenumbrüche werden in C++-Code ignoriert.

IGNORE_XML:

```
(\s \// spaces
|<!--([^-]|-+[^->]|->)*-+->)+ // block comment
```

XML ignoriert Blockkommentare, Leerzeichen und Zeilenumbrüche.

10.1.4.8 Zeilenumbruch

EOL: `\r?\n`

Diese Definition passt auf alle WINDOWS und UNIX Zeilenumbrüche, falls '\r' und '\n' nicht in den Projektoptionen als auszulassen gesetzt sind.

Im Editor werden alle Zeilen durch "\r\n" umgebrochen, auch, wenn die Quelldatei nur '\n' Umbrüche enthält

Als vordefiniertes Token kann EOL über ein Popup-Menü in ein Projekt eingefügt werden.

10.1.4.9 Binäre Null

NULL: `\x00`

NULL ist als Zeichen mit dem Wert Null definiert. Die Verwendung dieses Tokens ist nur sinnvoll, wenn binäre Dateien geparst werden. In Textdateien kann es nicht vorkommen. Dort markiert es das Ende der Datei EOF.

Als vordefiniertes Token kann *NULL* über ein Popup-Menü in ein Projekt eingefügt werden.

10.1.4.10 Adressen

Die folgenden vordefinierten Token können über ein Popup-Menü in ein Projekt eingefügt werden.

EMAIL:

```
[\w\.-]+ \// local part
@ \
([\w-]+\.)+ \ // sub domains
[a-zA-Z]{2,4} // top level domain
```

Eine gewöhnliche E-Mail Adresse. Die vollständige Syntax einer E-Mail Adresse ist ziemlich komplex und läßt sich nicht vernünftig mit einem regulären Ausdruck erfassen.

IP_ADDRESS:

```
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\. \
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\. \
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\. \
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
```

IP Adresse : alle vier Nummern im Bereich 0..255

Die Adressausdrücke sind dem Artikel von Vasant Raj entnommen:

http://www.codeproject.com/useritems/Regular_Expressions.asp

Der Ausdruck für die E-Mail Adresse wurde vereinfacht.

10.1.4.11 Daten-Feld

Wenn im Popup-Menü der vordefinierten Token der Eintrag: Daten-Feld aktiviert wird, erscheint der Zeichenmengen-Berechner, mit dem Ausdrücke erzeugt werden können, die z.B. Datenfelder einer Datenbank-Tabelle erkennen können.

Felder einer Datenbank-Tabelle sind häufig durch eine bestimmte Feldbreite: eine Anzahl von Zeichen definiert. Die Zeichen, die innerhalb des Feldes verwendet werden dürfen ist ebenfalls eingeschränkt. Z.B. kann ein Feld für zehn alphanumerische Zeichen definiert sein:

```
[[:alnum:]]{10,10}
```

10.1.5 Platzhalter

Platzhalter-Token erhalten ihre Bedeutung erst aus dem Eingabetext. Textabschnitte - meist Worte -,die zunächst durch einen allgemeinen regulären Ausdruck erkannt werden, können einem Platzhalter-Token zugeordnet werden. Kommt der gleiche Textteil an späterer Stelle im Eingabetext erneut vor, so kann er nun durch das bisherige Platzhalter-Token erkannt werden.
Dabei können einem Platzhalter beliebig viele literale Ausdrücke zugewiesen werden. Sind ihm mehrere Ausdrücke zugewiesen, so bilden sie eine Reihe von Alternativen, von denen jeweils nur eine an einer Textstelle passen kann.

Dynamische Token werden nur getestet, wenn sie als eine mögliche Alternative erwartet werden.

Unabhängig von den Projekteinstellungen wird bei Platzhalter-Token stets zwischen Groß- und Kleinschreibung wird unterschieden und die Wortgrenzen-Option ist gesetzt.

Ein Platzhalter-Token wird durch den Ausdruck:

{DYNAMIC}

definiert. Der Ausdruck *{DYNAMIC}* ist ein Schlüsselwort und wird vom TextTransformer **nicht** als Makro interpretiert.

Die Zuordnung der literalen Ausdrücke erfolgt in semantischen Aktionen mittels des Befehls: *AddToken*.

Beispiel:

Variablen, die in einem Programm-Code zunächst mit bestimmten Typ deklariert werden, können einem Platzhalter für diesen Typ zugewiesen werden. Im Definitionsteil des Codes ist dann der Typ der Variablen bekannt.

```
ID ::=  \w+
INTEGER ::= {DYNAMIC}

"int"
(
```

```
    ID
    {{
    AddToken( xState.str(), "INTEGER" );
    // ordnet INTEGER den gefundenen Bezeichner zu
    }}
    ","
)*

";"

INTEGER
```

Werden durch *ID* der nacheinander die Namen : *i1, i2* und *i3* erkannt, so kann an späterer Stelle mit *INTEGER i1* oder *i2* oder *i3* erkannt werden. *INTEGER* hat jetzt die Bedeutung eines Tokens, das definiert wäre als: i1|i2|i3

Anmerkung: Die Erweiterung eines Platzhalter-Tokens durch zusätzliche Literale hat **keine modifizierende Wirkung auf** die Erkennung eines vorausgehenden **SKIP-**Symbols.

10.2 Produktionen

Um einen Eingabetext umzuformen muss er gemäß seiner Syntax analysiert werden. Diese Analyse erfolgt anhand von Regeln ("Produktionen"), die die Syntax des Textes beschreiben. In diese Regeln eingebettet sind die Anweisungen, die bestimmen, wie aus dem analysierten Text ein neuer Text konstruiert wird.

Eine Produktion kann als eine Funktion betrachtet werden - genauer als eine Spezifikation zur Erzeugung einer Funktion -, **die einen Textabschnitt parsen kann**, falls er die in der Produktion definierte Struktur aufweist. Dieser Funktion können Parameter übergeben werden und sie kann einen Wert zurückliefern. Sie hat ihren eigenen Sichtbarkeitsbereich für Variablen und Konstanten. Innerhalb einer Produktion können andere Produktionen wie Funktionen aufgerufen werden. Die aufgerufenen Produktionen parsen wiederum Unterabschnitte des Textes, der von der aufrufenden Produktion erfasst wird.

Die Definition einer Produktion erfolgt innerhalb eines Formulars.

10.2.1 Eingabemaske für eine Produktion

Die Eingabemaske der Regelskripte umfasst folgende Felder:

Name:	eindeutigen Name
Rückgabetyp:	C++-Variablentyp
Parameter:	C++-Parameterdeklaration
Kommentar:	beliebiger Kommentartext
Text:	Skripttext der Produktion

Die Angabe von Namen und Text sind erforderlich. Erst wenn diese Felder gefüllt sind, ist es möglich

das Skript in die Verwaltung zu übernehmen oder einen Kommentar zu schreiben.

10.2.1.1 Name

Jeder Produktion muss ein Name gegeben werden. Die Namen der Produktionen werden in den Skripten anderer Produktionen verwendet, um die im Text des ersten Skripts beschriebene Struktur zu bezeichnen.
Der Name kann aus den Buchstaben des Alphabets, Ziffern und Unterstrichen '_' gebildet sein, wobei der Unterstrich nicht an erster Stelle des Namens stehen darf.

Beispiele.: Const_declaration, line

Jeder neue Name muss sich in mindestens einem Zeichen von allen anderen Namen der in der Verwaltung enthaltenen Produktionen, Token, Funktionen und Variablen unterscheiden. Es ist notwendig, dass sich nicht nur die Namen der Produktionen untereinander und die Namen der Token untereinander unterscheiden, sondern auch alle Namen der Produktionen müssen von denen der Token verschieden sein. das gleiche gile für die benutzerdefinierten Funktionen und Variablen.

10.2.1.2 Rückgabetyp

In das Feld für den Rückgabetyp muss genau dann eine Typenbezeichnung eingegeben werden, wenn es eine entsprechende Return-Anweisung in mindestens einer der semantischen Aktionen der Produktion gibt. Andernfalls wird automatisch eine Prozedur des Typs void erzeugt.
Es ist aber durch entsprechende Klammerung möglich gezielt zu bestimmen, ob ein Rückgabetyp im erzeugten Sourcecode, im internen Interpreter oder in beiden Verwendung finden soll.
Beispielsweise wäre es möglich einen String für die TETRA-interne Verwendung und einen Zeiger für den erzeugten Sourcecode zurückzugeben:

```
{-str-} {_CToken*_}
```

Ohne Klammern gilt für den Text des Rückgabetyps die Projektoption, die für die doppelt geschweifte Klammer "{{...}}" eingestellt ist.
Es ist darauf zu achten, dass jeweils die interpretierbaren und der exportierbaren Codeteile in sich stimmig bleiben. Nur für den interpretierbaren Teil kann der TextTransformer eine Typüberprüfung durchführen.

Wenn in Code, der nur für den Export bestimmt ist ein Rückgabetyp definiert ist, muss ein Default-Wert angegeben werden. Dies kann hier geschehen, indem ein Schrägstrich mit nachfolgendem Wert geschrieben wird. Z.B.:

```
{_  CProduktion*  _}/NULL
```

10.2.1.3 Parameterdeklaration

In das Parameterfeld können Parameterdeklarationen für die Produktion eingegeben werden, z.B.:

```
str& s, int i
```

Für eine solche Parameterdeklaration gilt die Projektoption, die für die doppelt geschweifte Klammer

"{{...}}" eingestellt ist.
Es ist aber durch entsprechende Klammerung einzelner Parameter auch möglich gezielt zu bestimmen, ob ein Parameter im erzeugten Sourcecode, im internen Interpreter oder in beiden Verwendung finden soll. Beispielsweise wäre es möglich den String nur für die TETRA-interne Verwendung und die Integer-Variable nur für den erzeugten Sourcecode zu definieren:

```
{-str& s-} {_int i}
```

Es ist darauf zu achten, dass die Texte der Deklarationen für jeweils einen Verwendungsbereich hintereinander geschrieben eine in C++ gültige Liste von Parameterdeklarationen ergibt.

```
{-str& s-} {_int i_} {-double d-}
```

ergäbe für den Interpreter: str& s double d. Es fehlt also ein Komma zur Trennung der Parameter. Richtig wären:

```
{-str& s-} {_int i_} {-, double d-}  oder
{-str& s,-} {_int i_} {-double d-} oder
{-str& s, double d-} {_int i_}  oder
{_int i_} {-str& s, double d-}
```

Die Syntax der Parameterdeklarationen wird vom TextTransformer nur geprüft, soweit sie für den Interpreter bestimmt sind. Die allein für den Export vorgesehenen Deklarationen werden schlicht in den generierten Code kopiert. Erst, wenn der erzeugten Code in ein C++-Programm eingebunden wird, wird er bei dessen Compilierung geprüft.

10.2.1.4 Kommentar

Ein Kommentartext zur Produktion wird in dem gelblich hinterlegten Feld dargestellt.

Dieses Feld wird temporär auch zur Anzeige von Fehlemeldungen etc. verwendet. Um den Kommentar zu ändern muss der Kommentarschalter betätigt werden, der eine gesonderte Dialogbox zum Edieren des Kommentartextes öffnet.

10.2.1.5 Text

Der Text einer Produktion definiert eine grammatische Regel. Die Syntax für die Definition ist an die Syntax der regulären Ausdrücke angelehnt. Einige der gleichen Metazeichen werden benutzt und einige neue Schlüsselworte werden eingeführt.

Die folgende Liste führt alle verwendeten Metazeichen an, und wofür sie stehen :

| die Trennung von Alternativen

(...) eine Gruppierung von Ausdrücken
? eine optionale Gruppierung

* die optionale beliebig häufige Wiederholung einer Gruppierung
+ die einmalige oder beliebig häufige Wiederholung einer Gruppierung

[…] Aufrufparameter einer Produktion

{-...-} semantische Interpreteraktionen
{_..._} semantische Aktionen zum Export in generierten Code
{=...=} semantische Aktionen sowohl für den Interpreter als auch zum Export
{{...}} semantische Aktionen, die gemäß den Projekteinstellungen für den Interpreter oder den Export bestimmt sind

// leitet Zeilenkommentare ein

Zudem gibt es die Schlüsselworte:

BREAK zum Verlassen einer Erkennungsschleife
EXIT zum Beenden des Parsens
EOF Ende des Quelltextes (end of file)
SKIP Sprung zu den Nachfolgetoken der Alternativen
IF...ELSE...END Bedingte Ausführung
WHILE...END Bedingte Ausführung

10.2.2 Elemente

So wie in den Token Einzelzeichen und Zeichenmengen verknüpft sind, so sind die Elemente der Produktionen die Token. Bestimmte regelmäßige Folgen oder Alternativen von Tokenfolgen lassen sich als eine Parserregel (Produktion) formulieren.
Definitionen von Produktionen beruhen auf drei Arten von Elementen:

1a) unmittelbar in der Produktion definierte literale Token
1b) auf der Tokenseite definierte und somit benannte Token
2. anderen Produktionen

Anmerkung: Auf den Text, der von einem Token erkannt wurde, kann durch xState.str() zugegriffen werden.

zu 1a) **unmittelbar in der Produktion definierte literale Token**

Einfache Worte der natürlichen Sprache oder Schlüsselworte formalisierter Sprachen, Satzzeichen, Operatoren etc., müssen aber nicht gesondert auf der TETRA-Tokenseite definiert werden. Diese literalen Token können direkt innerhalb einer Produktion definiert werden, indem man sie in Anführungszeichen setzt.

Beispiele:

"TETRA" passt auf: TETRA
";" passt auf: ;

Soll das Anführungszeichen selbst als Bestandteil eines literalen Tokens verwendet werden, so

muss ihm ein Backslash '\'-Zeichen vorangestellt werden, um es von den umschließenden Anführungszeichen zu unterscheiden. Ebenso muss dem Backslash ein Backslash vorangestellt werden.

"\""	passt auf: "
"\\"	passt auf: \

zu 1b) **auf der Tokenseite definierte und somit benannte Token**

Tokendefinitionen wurden weiter oben ausführlich dargestellt. Bei der Definition von Produktionen werden die Namen der bereits definierten Token verwendet.

zu 2. **andere Produktionen**

Die Definition einer Produktion kann auf Definitionen anderer Produktionen beruhen, indem deren Namen verwendet werden.

Beispiel:

Production1 ::= "hallo" | "tschüss"
Production2 ::= "Welt"
Production3 ::= Production1 Production2

10.2.3 Verkettung

Folgen zwei oder mehr Symbole in einer Regel direkt aufeinander, so bilden sie eine Kette. Beispielsweise die Folge:

```
"TETRA" "macht" "Spaß"
```

ist eine Verkettung der drei Token: "TETRA", "macht" und "Spaß".

Ebenso wäre eine Folge

```
"TETRA" "macht" emotion
```

denkbar. Hierin steht **emotion** entweder für ein auf der Tokenseite definiertes Token oder für eine andere Produktion. Welcher der beiden Fälle zutrifft entscheidet TETRA, indem es sowohl die Liste der Token als auch die Liste der Produktionen nach dem Namen **emotion** durchsucht. Damit die Entscheidung eindeutig zu fällen ist, müssen die Namen aller Tokendefinitionen und aller Regeldefinitionen sämtlich voneinander verschieden sein.

Semantische Aktionen sind ebenfalls Teil der eben beschriebenen Ketten.

Die Verkettung hat die höchste Priorität aller Operatoren einer Produktion.

10.2.4 Alternativen

Alternativen werden durch das Pipe-Zeichen "|" voneinander getrennt.

Beispiele:

"alles" | "nichts" steht für ein alternatives Vorkommen im Text von "alles" oder "nichts".

text | **zahl** steht für die alternativen Produktionen **text** und **zahl**

Ein Sonderfall sind **leere Alternativen**. Leere Alternativen können sinnvoll sein, wenn eine bestimmte Aktion ausgeführt werden soll, falls keine der anderen Alternativen passt. Beispielsweise kann dann eine Fehlermeldung ausgegeben werden:

```
"alles" | "nichts" | {{out << "Fehler"; }}
```

Anmerkung:
Die Anfängermenge einer leeren Alternative enthält das spezielle Token **EPS** mit der Nummer 2. n dem speziellen Fall, dass auf die leere Alternative kein Token mehr in der Grammatik folgt, wird im Tokenfenster des Debuggers das erwartete Token mit "EPS" bezeichnet.

Es ist darauf zu achten, dass nicht **versehentlich eine leere Alternative gebildet** wird, wie es in folgenden Produktion der Fall ist:

```
X ::= (
       | "empty"
       | "size"
       | "clear"
      )
```

Vor jedem der angeführten Token steht das Zeichen '|' zur Alternativenbildung. Vor dem ersten dieser Zeichen steht aber nichts, wozu die Alternative gebildet würde. Äquivalent hätte formuliert werden können:

```
X ::= (
         "empty"
       | "size"
       | "clear"
       |
      )
```

Um das versehentliche Erzeugen leerer Alternativen zu verhindern, werden die Formulierungen der letzen beiden Beispiele als syntaktisch falsch gewertet und verursachen Fehlemeldungen. Soll bei nicht passenden Alternativen ein Parserfehler vermieden werden, ohne dass eine semantische Aktion ausgeführt wird, so kann entweder die Gesamtheit der Alternativen optional gemacht werden:

```
( "alles" | "nichts" )?
```

oder es kann eine leere Alternative mit einer leeren Aktion gebildet werden:

```
"alles" | "nichts" | {{ /*Empty*/ }}
```

Anmerkung: Leere Alternativen sind löschbar.

10.2.5 Gruppierung

Durch die Klammern '(' und ')' können Ausdrücke innerhalb der Definition einer Produktion gruppiert werden.
Dadurch kann der Vorrang der Verkettung aufgehoben werden, wenn Gruppen von Alternativen zu bilden sind.

Beispiel:

("a" | "b") ("c" | "d")	passt auf	"a c", "a d", "b c" und "b d"
"a" | "b" "c" | "d"	passt auf	"a", "b c" und "d"

Wird der Vorrang der Verkettung außer Acht gelassen,so führt dies häufig zu verblüffenden Ergebnissen insbesondere dann, wenn semantische Aktionen beteiligt sind.

Beispiel:

```
{{str s;}}
"a"
{{s = "a"; }}
| "b"
{{s = "b"; }}
```

ergibt für die letzte Zeile die Fehlermeldung: Unbekannter Bezeicher : s
Implizit wurden die ersten drei Zeilen geklammert, so dass nur in diesem Bereich die Deklaration von str s gültig ist. Korrekt wäre zu schreiben:

```
{{str s;}}
(
"a"
{{s = "a"; }}
| "b"
{{s = "b"; }}
)
```

Ein weiteres Beispiel hierfür ist die in der Einleitung beschriebene Inner-Produktion, bei der eine Klammerung um "b" | "c" erforderlich ist .

10.2.6 Wiederholungen

Die Syntax für Wiederholungen ist analog zur entsprechenden Syntax der regulären Ausdrücke, nur dass die Operatoren auf ganze Token, Produktionen oder Gruppen derselben wirken.

+

Ein Token, eine Produktion oder Gruppe, die von einem Pluszeichen '+' gefolgt wird steht mit einer beliebig häufigen Wiederholung dieser Gruppierung im Text in Übereinstimmung, die Gruppierung muss dort aber mindestens einmal gefunden werden.

*****

Ein Token, eine Produktion oder Gruppe, die von einem Stern '*' gefolgt wird steht ebenfalls mit einer beliebig häufigen Wiederholung dieser Gruppierung im Text in Übereinstimmung, sie muss aber nicht im Text vorkommen. Letzterer Fall wäre quasi eine "null-maligen" Wiederholung.

?

Ein Token, eine Produktion oder Gruppe, die von einem Fragezeichen '?' gefolgt wird steht mit einem null- oder einem einmaligen Vorkommen im Text in Übereinstimmung. Es ist eine optional vorkommende Gruppierung.

{}

Es ist möglich eine minimale und eine maximale Anzahl von aufeinander folgenden Wiederholungen eines Musters explizit festzulegen. Hierzu dient der Operator "{}".
A{2} bedeutet, dass das Token oder die Produktion oder die Gruppe *A* exakt zwei mal wiederholt wird.
A{2,4} bedeutet, dass der *A* mindestens zwei und maximal vier mal hintereinander im Text konsumiert wird.
A{2,} *konsumiert mindestens* zwei *A* und beliebig viele weitere.

Achtung: wenn die Mindestanzahl der Muster im Text nicht gefunden wird, gibt es einen Fehler, wenn das Muster hingegen häufiger im Text aufeinander folgend vorkommt, als durch die Maximalzahl spezifiziert, gibt es nur dann einen Fehler, wenn das nächste Textmuster nicht anders erkannt wird.

Im Gegensatz zur entsprechenden Syntax bei regulären Ausdrücken darf {} Leerzeichen enthalten.

Anmerkung:
Die WHILE-Struktur bietet die Möglichkeit die Anzahl von Wiederholungen dynamisch zu bestimmen.

10.2.7 BREAK

Mit **BREAK** können Schleifen - (...)* oder (...)+ - abgebrochen werden. Stößt der Parser innerhalb einer Schleife auf ein **BREAK**-Symbol, so wird die Schleife verlassen und das Parsen wird mit den auf die Schleife folgenden Token und Regeln fortgesetzt.
Z.B.:

```
( "a" "b" "c"| "d" | BREAK)+  "e"
```

würde auf folgende Texte passen:

"a b c d a b c e"
"d d d e"
"e"

```
(A | BREAK)+  wäre äquivalent zu (A)*
```

Da mit **BREAK** stets ein Schleifendurchlauf abgebrochen wird, kann auf **BREAK** kein weiterer Knoten folgen, auch keine semantische Aktion. Soll eine solche mit **BREAK** verbunden werden, so muss sie vor dem **BREAK**-Symbol definiert werden:

```
( "a" "b" "c"| "d" | {{out << "break";}} BREAK)+  "e"
```

Das **BREAK**-Symbol muss in der gleichen Produktion stehen in der auch die Schleife definiert ist, die durch das **BREAK**-Symbol abgebrochen wird und das **BREAK**-Symbol hat nur innerhalb einer Schleife Bedeutung. Es ist also nicht möglich das obige Beispiel in zwei Produktionen aufzusplitten:

```
xxx ::=
( "a" "b" "c"| "d" | Break)+  "e"

Break =
{{out << "break";}} BREAK  // falsch
```

Die Verwendung des **BREAK**-Symbols ist in diesen genannten Aspekten analog zum Schlüsselwort "break" in C++. Tatsächlich wird im erzeugten Code **BREAK** durch "break" ersetzt.

Mittels des **BREAK**-Symbols lassen sich Textstrukturen analysieren, die die normale Top-Down-Analyse - z.B. Parser, die auf der reinen EBNF-Syntax beruhen - vor ein unlösbares Problem stellen. Ein Text könnte beispielsweise folgender Regel gemäß strukturiert sein:

```
(";" "a" )+ ";" "b"
```

Diese Regel ist syntaktisch einwandfrei, provoziert jedoch die Warnmeldung:

";" ist Anfänger und Nachfolger löschbarer Strukturen

In diesem Fall darf die Warnung nicht ignoriert werden. Das Parsen des Eingabetextes

"; a ; b"

z.B. führt zu einem Fehler. Nachdem "; a" erkannt wurde, gibt es einen Konflikt zwischen einer Fortsetzung mit einem erneuten Durchlauf der Schleife und einer Fortsetzung mit ";" "b". Der

TextTransformer entscheidet sich in solchen Fällen stets für die erste Alternative. Nach Erkennung des zweiten Semikolons wird also "a" erwartet und nicht das im Text stehende "b".
Hier kann das **BREAK**-Symbol helfen, indem die Regel umformuliert wird zu:

```
(";" ( "a" | BREAK ) )+ "b"
```

Der Eingabetext wird nun richtig erkannt. Wieder wird das zweite Semikon am Beginn des zweiten Durchlaufs der Schleife erkannt. Nun wird aber die **BREAK**-Alternative gewählt und damit wird die Schleife verlassen und das "b" im Text wird durch das der Schleife folgende "b" der Regel richtig erkannt.
Eine andere Umformulierung der ersten Regel hätte auch zu einer Erkennung des gesamten Textes geführt:

```
(";" ( "a" )? )+ "b"
```

Diese Regel würde jedoch auch zu Texten passen, die nicht in der ursprünglichen Intention lagen. Z.B. zu ";;;b"

10.2.8 EXIT

Trifft der Parser auf das Schlüsselwort **EXIT**, so wird die Textanalyse abgebrochen. Der Abbruch erfolgt aus die gleiche Weise, wie bei aktiviertem Interpreter durch die throw-Anweisung.

EXIT kann von dem weiteren Schlüsselwort **OK** gefolgt werden. Damit wird angezeigt, dass der Programmabbruch regulär ist. Ohne **OK** weist der Programmabbruch auf einen Fehler hin.

Da mit **EXIT** das Programm abgebrochen wird, kann auf **EXIT** bzw. **EXIT OK** kein weiterer Knoten folgen, auch keine semantische Aktion. Soll eine solche mit **EXIT** verbunden werden, so muss sie vor dem **EXIT**-Symbol definiert werden.

Beispiel:

```
{{ out << xState.FileName() << " fertig" << endl; }}
EXIT
OK
```

EXIT kann in Vorausschau-Parsern verwendet werden, um die Vorausschau zu beenden. Beim Gebrauch in Vorauschau-Parsern wird dieser regulär beendet ohne eine Ausnahme auszuwerfen.

In dem vom TextTransformer erzeugten C++-Code wird **EXIT** durch Auswurf einer Exception realisiert, wenn nicht UseExcept abgeschaltet ist und nicht gerade ein Unter- oder Vorausschau-Parser ausgeführt wird.

```
throw tetra::CTT_Exit("EXIT, true);
```

Diese Ausnahme wird im erzeugten Code nicht automatisch abgefangen.

10.2.9 EOF

EOF steht für "end of file", Dateiende. (Genauer müsste es heißen: Ende des Eingabestrings.) Dieses spezielle Token wird im Normalfall automatisch erzeugt: Es gehört stets zur Nachfolgermenge der Startregel und zu den Nachfolgermengen der dem Ende der Startregel vorausgehenden löschbaren Strukturen. Ist die Startregel selbst löschbar, so gehört **EOF** auch zur Anfängermenge dieser Regel.

Es ist auch möglich explizit **EOF** innerhalb einer Produktion zu verwenden. Hinter dem Dateiende **EOF** kann kein Token mehr folgen. Bis zum Ende des Programms dürfen daher entweder nur löschbare Strukturen oder *EOF*-Alternativen folgen, oder das Programm sollte mit **EXIT** abgebrochen werden.

Beispiel.:

```
"a"
("b" | EOF EXIT)
"c"
```

Hinweis: Mittels des *EOF*-Symbols sind die verkürzten Ausgabe- und Zuweisungs-Anweisungen (z.B.: {{out << }}) für Rückgabewerte nachfolgender Regeln in den Interpreter integriert worden.

10.2.10 ANY

Das Symbol ***ANY*** bezeichnet ein einzelnes Token aus der Menge aller im Projekt vorkommenden Token, das nicht alternativ zu diesem *ANY* Symbol in der gleichen Produktion steht. Es kann verwendet werden, um auf einfache Weise beliebigen Text zu parsen. Zum Beispiel kann der Inhalt einer exportierbaren Aktion übergangen werden durch:

```
"{_" ANY* "_}"
```

In diesem Beispiel ist das schließende "_}" eine implizite Alternative des wiederholten *ANY* Symbols. Das bedeutet, dass *ANY* auf alle Token passt außer "_}".

ANY erkennt nur Token, die irgendwo innerhalb der von der Startregel abhängigen Regeln mindestens einmal explizit vorkommen: Dieser Punkt gilt für jedes Parsersystem gesondert. Wenn in dem Regelsystem die literalen Token "int", "=" und ";" vorkommen und außerdem ein ID-Token für Bezeichner und ein NUMBER-Token für Nummern, so wird mit dem obigen Beispiel der Text

```
{_ int i = 3; _}
```

erkannt als

```
"{_" ANY ANY ANY ANY ANY "_}"
```

wobei *ANY* jeweils die entsprechenden Token vertritt

```
"{_" "int" ID "=" NUMBER ";" "_}"
```

Wenn das literale Token "int" sonst nicht in den Regeln vorkommt, ist das auch kein Problem, weil

"int" dann als Bezeichner erkannt wird. Sollte aber *NUMBER* sonst nicht im Regelsystem verwendet werden, so muss es hier explizit als Alternative eingefügt werden:

```
"{_" (ANY | NUMBER )* "_}"
```

Andernfalls würde der Text nicht geparst.

10.2.10.1 Optionen

Dieser Abschnitt ist nur für Spezialisten. Dem normalen Benutzer wird empfohlen, die Defaulteinstellung "keine Fehlschlagsalternative für ANY" in den Projektoptionen zu belassen und diesen Abschnitt zu überspringen.

In den Projektoptionen kann gewählt werden, ob der Kontext der Produktion in der ein ANY-Symbol steht bei der Berechnung seiner Tokenmenge berücksichtigt wird oder nicht.

1. keine Fehlschlagsalternative

Gemäß dieser Option werden die Alternativen in der eine Produktion steht, die ein die *ANY*-Symbol verwendet, bei der Berechnung der Tokenmenge dieses Symbols berücksichtigt. Beispiel:

```
Produktion1  ::= ANY | "a"
Produktion2  ::=  "b"
Produktion3  ::=  Produktion1  |  Produktion2
```

Dadurch, dass *Produktion2* eine Alternative zu *Produktion1* ist, wird "b" zu einer Alternative von *ANY* . *ANY* erkennt somit alle Token außer "a" und "b".

Diese Option ist für neue Projekte voreingestellt, da sie den Intuitionen des Anwenders entsprechen dürfte.

Allerdings gibt es in einem speziellen Fall ein Problem. Steht das ANY-Symbol in einer löschbaren Struktur am Ende einer Produktion und hat diese in verschiedenen Kontexten verschiedene Nachfolger, kann es zu einem unerwarteten Verhalten kommen.

```
Any ::= ANY+
Production ::= "a" Any "b" | "c" Any "d"
```

Der Text "a d b" wird in diesem Fall nicht geparst. Die erste Alternative ließe das erwarten, aber wegen der zweiten wird "d" aus der Menge der von *ANY* erkannten Token ausgeschlossen.
In solchen Fällen generiert der TextTransformer eine Warmeldung.

2. Fehlschlagsalternativen

Einfacher und daher etwas schneller ist die Berechnung der von *ANY* erkannten Tokenmenge, wenn der Kontext nicht berücksichtigt wird. Das hat aber den gravierenden Nachteil, dass z.B. die obige *Produktion3* nicht kompiliert. ANY erkennt dann alle Token außer "a" und steht somit in Konflikt mit *Produktion2*.
Sinnvoll kann diese Option sein, wenn Kompatibilität mit Coco/R-Projekten gewahrt werden soll, da ANY dort auf diese Weise berechnet wird.

10.2.11 SKIP

Mittels des **SKIP**-Symbols können Abschnitte des Quelltextes für die keine expliziten Regeln existieren beim Parsen zu übersprungen werden.

Anmerkung: Der übersprungene Text ist durch xState.str() oder trim_right_copy(xState.str()) zugänglich.

Das Schlüsselwort **SKIP** steht für ein komplexes Symbol, dessen genaue Bedeutung vom jeweiligen Kontext innerhalb eines TETRA-Programms abhängt. Die Bedeutung hängt ab:

1. von den jeweiligen Alternativen

Eine *SKIP*-Alternative wird dann ausgewählt, wenn keine der anderen Alternativen auf den aktuellen Text passt. In einer Produktion:

```
(
    "}"
  | "]"
  | SKIP
)*
```

wird nur dann die *SKIP*-Alternative gewählt, wenn an der aktuellen Textposition keine der schließenden Klammern "}" oder "]" steht.

2. von den jeweiligen möglichen Nachfolgern

Das SKIP-Symbol passt auf den Text, der an der aktuellen Position beginnt und sich bis zu derjenigen Position erstreckt, ab der der Text zu einem Nachfolgesymbol passt. Gibt es mehrere Nachfolgesymbole, die auf nachfolgende Textabschnitte passen, so wird dasjenige gewählt, dessen Übereinstimmung mit dem Text an der kleinsten Position beginnt.

Lautet beispielsweise der aktuelle Eingabetext:

```
param1, param2 ] }   ...
```

und die Regel

```
SKIP
(
    "]"
  | "}"
)
```

so wird durch den *SKIP* der Text "param1, param2 " erkannt, da "]" auf diesen Text unmittelbar folgt, während "}" erst an späterer Stelle folgt.
Die Beispielsregel unter Punkt eins wird dasselbe Resultat haben. Beim ersten Durchlauf der Schleife passt das *SKIP*-Symbol auf den Text und beim zweiten Durchlauf passt "]". Beim dritten Durchlauf wird dann auch "}" erkannt.
Allerdings ist zu beachten, dass das, was durch SKIP in dieser Regel erkannt wird, auch von den Nachfolgern der Schleife selbst abhängt. Im folgenden Kontext:

```
Startregel  ::=  Regel1  Regel2
```

```
Regel1  ::=
(
     "]"
   | "}"
   | SKIP
)*

Regel2  ::=  "param2"
```

würde lediglich der Abschnitt: "param1, " erkannt.

Ergänzende Erläuterungen

Mögliche Konflikte werden je nach gesetzten Optionen verschieden behandelt.

a) Isolation von SKIP und ANY

Die Vorkommen des Schlüsselworts "SKIP" müssen voneinander isoliert sein, d.h. ein *SKIP* darf keinen zweites *SKIP* als Alternative haben und darf kein zweites *SKIP* als unmittelbaren Nachfolger haben. Folgendes ist also nicht erlaubt:

```
(SKIP | ...) | (SKIP | ...)  // falsch
```

oder

```
(SKIP | ...) ( SKIP | ...)  // falsch
```

Ebenso darf ANY nicht unmittelbar auf *SKIP* folgen.

```
(SKIP | ...) ( ANY | ...)  // falsch
```

b) SKIP-Wiederholungen

Die folgende Fälle unterscheiden sich nicht, sie resultieren in einem gleichen Verhalten des Parsers:

```
SKIP?
SKIP*
```

Ein passendes Nachfolgesymbol muss sich entweder an der aktuellen oder an einer späteren Position im Text befinden. Im letzteren Fall wird *SKIP* einmal ausgeführt.
Der Fall

```
SKIP+
```

unterscheidet sich von den vorhergehenden darin, dass ein Nachfolgesymbol nicht an der aktuellen Position vorhanden sein darf, an einer späteren aber vorkommen muss; aber auch hier wird die Wiederholung auf eine einmalige Ausführung reduziert.

c) Konsumation der auszulassenden Zeichen

Die Funktion des SKIP Symbols hängt nicht von den Projektoptionen ab! Z.B. wird SKIP EOL auch dann das Zeilenende erkennen, wenn '\r' und '\n' zu den auszulassenden Zeichen gehört. Während xState.str() nach Erkennung eines normalen Tokens einen Textabschnitt liefert, der der vor den nachfolgenden auszulassenden Zeichen abbricht, ist dies bei einem durch SKIP erkannten Textabschnitt nicht der Fall. Wenn die auszulassenden Zeichen Leerzeichen sind, erhält man das entsprechende Resultat durch:

```
trim_right_copy( xState.str())
```

d) kein dynamisches SKIP

Die Menge der Token, die auf SKIP folgen können ist **nicht dynamisch veränderbar**. Wenn ein Platzhalter-Token, das auf SKIP folgt, durch ein Literal erweitert wird, wird hat Erweiterung keine Auswirkungen auf die Erkennungen durch das SKIP-Symbol.

e) Namen der SKIP-Knoten

Anmerkung: Der Name eines Skip-Knotens wird gebildet aus "SKIP" und der Nummer, die den Knoten in den Anfängermengen repräsentiert. Also, z.B. SKIP12 ist ein Skipknoten, der in der Anfängermenge des übergeordneten Knotens als Knoten mit der Nummer 12 registriert ist.

10.2.11.1 Optionen

Es gibt drei Optionen für das *SKIP*-Symbol, die die Behandlung möglicher Konflikte regeln. Zwei davon können in den Projektoptionen voreingestellt werden. Soll eine nicht voreingestellte Option verwendet werden, so kann dies durch Anhängen eines Parameters an "SKIP" ausgedrückt werden. Z.B.

```
SKIP[ F ]
```

1. keine Fehlschlagsalternative

Wenn in den Projektoptionen die Verwendung globaler Scanner eingestellt ist, kann es passieren, dass an der aktuellen Position des Quelltextes ein Token erkannt wird, das in der Grammatik nicht erwartet wird. Beispielsweise ist in sehr vielen Projekten ist ein regulärer Ausdruck zu Erkennung von Bezeichnern definiert. Wird nun mit

```
SKIP "Welt"
```

der folgende Text geparst:

```
Hallo Welt
```

so würde "Hallo" als Bezeichner interpretiert. Bei strikter Anwendung des Scanner-Algorithmus würde SKIP dann nicht getestet und es würde eine Fehlermeldung erzeugt. Wenn aber keine Fehlschlagsalternativen zugelassen sind, wird "Hallo" durch SKIP übersprungen und somit der gesamte Text korrekt geparst.
Innerhalb einer Produktion kann diese Option mit dem Parameter NFA (no failure alternative) explizit gesetzt werden:

```
SKIP[ NFA ]
```

Allerdings gibt es bei dieser Option ein **Problem** mit **verdeckten Alternativen**: Beispiel:

```
( ID ";"* )* SKIP
```

Beginnt ein Text mit einem Semikolon, so wird es mit SKIP übersprungen. Beginnt der Text mit einem Bezeichner, auf den ein Semikolon folgt, so wird der Bezeichner korrekt erkannt. Das Semikolon wird aber wiederum übersprungen. Es wird zwar zunächst erkannt, aber nicht als Alternative zu SKIP gewertet.
In diesem Fall wäre die strikte Option 3 (s.u.) für SKIP angebracht.

2. Sprungziele an der aktuellen Position zulassen // experimentell

Ein ähnlicher Konflikt wie eben beschrieben entsteht, wenn schon an der aktuellen Position des Quelltextes ein Token erkannt wird, das als Ziel des SKIP-Knotens gesetzt ist.

```
SKIP ID
```

Das Wort "Hallo" im Text:

```
Hallo Welt
```

wird dann bereits als Bezeichner *ID* erkannt. Die oben beschriebene Option keine Fehlschlagsalternativen zuzulassen greift hier nicht, da die Sprungziele von SKIP nicht zu diesen Alternativen hinzugerechnet werden. Würden sie hinzugerechnet, wäre ein Ausdruck wie

```
SKIP? ID
```

sinnlos. SKIP würde immer auf den Text passen (abgesehen vom Textende, bei dem kein Bezeichner mehr folgt). Es ist also zu empfehlen, die Grammatik entsprechend abzuändern: Mit

```
SKIP? ID+
```

wird "Hallo Welt" korrekt geparst. Experimentell gibt es die NF-Option (no failure)

```
SKIP[ NF ] //  experimentell
```

Möglicherweise wird diese Option in zukünftigen TETRA-Versionen nicht mehr verfügbar sein, es sei denn sie wird von Benutzern ausdrücklich gewünscht.

3. strikte Fehlererzeugung

Bis zu TextTransformer 1.5.0 wurde in den oben dargestellten Fällen immer Fehler erzeugt. Diese Option zwingt den Programmierer dazu, Konfliktfälle explizit zu bedenken und zu behandeln. Die Möglichkeit *SKIP* optional zu machen wurde bereits angeführt. Ein andere Möglichkeit wäre:

```
( SKIP | ID ) ID
```

Strikte Fehlererzeugung kann in einer Produktion mit dem F-Parameter gesetzt werden:

```
SKIP[ F ]
```

Anmerkung
Nochmals sei darauf hingewiesen, dass die genannten Konfliktmöglichkeiten aus der Einstellung resultieren, globale Scanner zu verwenden. Ein Konflikt mit Literalen kann sich nur ergeben, wenn auch die Option "alle Literale testen" gesetzt ist und es ein Literal als unmittelbare Alternative zu SKIP gibt. Andernfalls wird auf Literale überhaupt nicht getestet.

10.2.12 IF...ELSE...END

In einer IF...ELSE...END Struktur werden Alternativen zugelassen, die an anderer Stelle einen LL(1)-Konflikt ergeben würden. Der Fortgang des Parsens wird hier nicht nur durch das nächstfolgende Token bestimmt, sondern wird auch über Prädikate gesteuert; z.B. einer Vorausschau im aktuellen Text (s. Anmerkung unten)

Genauer hat diese Struktur die Form:

```
IF( boolscher Ausdruck )
  If-Zweig
ELSE
  Else-Zweig
END
```

IF- und ELSE-Zweig sind hier beliebige Verkettunngen oder Gruppierungen von Token und semantischen Aktionen. Der ELSE-Zweig ist optional, so dass auch eine einfache IF-Abfrage möglich ist:

```
IF( boolscher Ausdruck )
  If-Zweig
END
```

Der boolsche Ausdruck wird nur ausgewertet, wenn eines der Token aus der Anfängermenge des IF-Zweiges erwartet wird. Kann der IF-Zweig nicht mit dem nächsten Token beginnen, wird der ELSE-Zweig ausgeführt, unabhängig davon ob die IF-Bedingung zutrifft oder nicht. Falls es keinen ELSE-Zweig gibt, ist die Struktur löschbar.

Der boolscher Ausdruck ist immer zugleich interpretierbar und exportierbar.

Beispiele:

Diese einfache Struktur kann z.B. verwendet werden, um den Konflikt in den folgenden Regeln aufzulösen:

```
Declaration      ::= Type ( IdentEqual )? QualIdent ";"
IdentEqual       ::= Ident "="
QualIdent        ::= Ident ( "." Ident )*

// z.B.: "int i = xState.itg;" oder "int i;"
```

Da sowohl *IdentEqual* als auch *QualIdent* mit *Ident* beginnen besteht hier ein LL(1)-Konflikt. Er lässt sich entweder durch Ausklammerung von *Ident* auflösen:

```
Declaration ::=
Type Ident
(
  ( "." Ident )*
| "=" QualIdent
)
";"
```

oder es kann mittels der *IdentEqual* als Vorausschau-Produktion geschrieben werden:

```
Declaration ::=
Type
IF ( IdentEqual() )
     IdentEqual
END
QualIdent ";"
```

Es gibt auch LL(1)-Konflikte, die nicht so einfach oder überhaupt nicht auflösbar sind. Hier muss die IF...ELSE...END Struktur verwendet werden.

Als boolscher Ausdruck kann auch schlicht eine Klassen-Variable dienen.

```
IF ( m_bProfile )
(
  {{ double start = clock_sec(); }}
  Production
  {{ out << clock_sec() - start << " s" << endl; }}
)
ELSE
  Production
END
```

1. Anmerkung:

Die folgende Struktur ist **unendlichen Schleife**, wenn die Bedingung falsch ist.

```
(
  IF(Bedingung)
     Produktion
  END
)*
```

Das zur Anfängermenge von *Produktion* gehörende erwartete Token wird während eines

Schleifendurchlaufs nicht konsumiert. Da dieses Token auch zur Anfängermenge der Schleife gehört, wird diese immer wieder erneut ausgeführt.

Stattdessen sollte geschrieben werden:

```
WHILE(Bedingung)
  Produktion
```

oder

```
(
  IF( Bedingung )
     Produktion
  ELSE
    BREAK
  END
)*
```

Das IF-Konstrukt des TextTransformers ist insofern nicht zu vergleichen mit dem IF-Konstrukt von Coco/R, bei dem die Bedingung vor die Schleife gesetzt wird.

2. Anmerkung:

Wenn einer der Zweige löschbar ist, wird die **gesamte Struktur** als **löschbar** betrachtet. Das kann eine **unerwartete Konsequenz** haben. Egal, was in folgender Struktur die Bedingung ergibt, wird immer das Token "d" erkannt, wenn es als nächstes im Text steht. Auch, wenn die Bedingung nicht erfüllt ist, ergibt sich kein Fehler, wenn im Text nicht 'c' sondern 'd' folgt.

```
IF ( Bedingung )
  "a"?
ELSE
  "c"
END
"d"
```

10.2.13 WHILE...END

Ebenso wie bei der IF-Struktur wird bei der WHILE...END Struktur durch einen boolschen Ausdruck entschieden, ob eine Verweigung ausgeführt wird.

Genauer hat diese Struktur die Form:

```
WHILE( boolscher Ausdruck )
  While-Zweig
END
```

Der While-Zweig ist beliebige Verkettung oder Gruppierung von Token und semantischen Aktionen.

Der boolscher Ausdruck ist immer zugleich interpretierbar und exportierbar.

Beispiel

```
NUMBER
{{
int i = 0, iCount = xState.itg();
}}
WHILE(i < iCount)
ReadData  {{ i++; }}
END
```

Zunächst wird die Anzahl der nachfolgenden Datensätze gelesen und dann mit *ReadData* die Datensätze selbst.

Beispiel:

```
EmptyBracket ::= "[" "]"
NonEmptyBracket ::= "[" IDENT "]"

WHILE ( EmptyBracket() )
   EmptyBracket {{ iEmptyBracketsCount++; }}
END
NonEmptyBracket
```

Die Produktion *EmptyBracket* wird hier zugleich zum Parsen und zur Vorausschau verwendet. Die leeren Klammern bis zur ersten nicht leeren Klammer werden gezählt.

10.2.14 Aktionen

Vor und nach der Erkennung eines Tokens oder dem Aufruf einer Produktion können sogenannte semantische Aktionen ausgeführt werden. (In sehr speziellen Fällen ist dies auch während der Erkennung möglich). Diese Aktionen dienen zumeist der Verarbeitung des zuletzt erkannten Textabschnitts. Die Anweisungen die in den Aktionen ausgeführt werden sollen, sind in den Text einer Produktion eingefügt. Zur Abhebung werden sie in spezielle Klammerpaare eingeschlossen. Die Anweisungen selbst werden in der Sprache C++ formuliert.
Da TETRA einerseits in der Lage ist einen Teil der C++-Befehle direkt zu interpretieren andererseits bei der Erzeugung von Sourcecode auf den gesamten Reichtum von C++ zurückgreifen kann, muss es eine Möglichkeit geben, zu bestimmen, ob eine C++-Anweisung für den internen Interpreter gedacht ist oder zum Export. Aus diesem Grunde gibt es mehrere Arten von Klammern in die der C++-Code innerhalb einer Produktion eingebettet werden kann.

{-...-}	semantische Interpreteraktionen
{_..._}	semantische Aktionen zum Export
{=...=}	semantische Aktionen sowohl für den Interpreter als auch zum Export
{{...}}	semantische Aktionen, die gemäß den Projekteinstellungen für den Interpreter oder den Export bestimmt sind

Der Teil der Sprache C++, den der TextTransformer interpretieren kann ist im nächsten Kapitel dargestellt: Interpreterbefehle.

Eine typische Aktion wäre z.B. das Kopieren des vom zuletzt erkannten Token abgedeckten Quelltextabschnitts in die Ausgabe:

```
{{out << xState.str();}}
```

Die Syntax der semantischen Aktionen wird vom TextTransformer nur geprüft, soweit sie für den Interpreter bestimmt sind. Die allein für den Export vorgesehenen Aktionen werden schlicht in den generierten Code kopiert. Erst, wenn der erzeugten Code in ein C++-Programm eingebunden wird, wird er bei dessen Compilierung geprüft.

Die verschiedenen Klammerungen sind nicht möglich innerhalb von Bedingungen der IF-Strukturen oder WHILE-Strukturen. Dieser Code ist immer beides: interpretierbar und exportierbar.

Das Projekt EditProds demonstriert den parallelen Einsatz von semantischem Code, der nur für den Interpreter bestimmt ist und semantischem Code für den Export.

10.2.14.1 Übergangsaktion

Neben den semantischen Aktionen gibt es auch eine Art sehr spezieller "syntaktischer" Aktionen. Eine Übergangsaktion wird ausgeführt, nachdem ein Token akzeptiert wurde und bevor das nächste ermittelt wurde. Dies ist der ideale Zeitpunkt zur Einfügung neuer dynamischer Token mit *AddToken*, da das neue Token dann bereits vor der Ermittlung des nächsten Tokens zur Verfügung steht. Übergangsaktionen können auch nötig sein, damit sich eine Produktion bei ihrer Verwendung zur Vorausschau im Text genauso verhält, wie sonst. Ein abweichendes Verhalten kann entstehen, wenn während des Parsens neue dynamische Token erzeugt werden, oder wenn der Textbereich geändert wird. Wenn eine solche Änderung als Übergangsaktion ausgeführt wird, wird sie temporär auch während der Vorausschau ausgeführt. Dennoch bleiben die Menge der dynamischen Token und der Textbereich vor und nach Ausführung der Vorausschau unverändert.

Als Übergangsaktionen kommen vor allem die Funktionen *AddToken*, *PushScope* und *PopScope* in Betracht. Als Übergangsaktionen werden sie ausgeführt, wenn sie verbunden mit einem Punkt an den Ausdruck für ein Token angehängt werden.

Beispiele:

```
ID.AddToken(xState.str(),    "CLASS")
ID[n].AddToken(xState.str(),    "CLASS")
ID[n].AddToken(xState.str(),    "CLASS",    "NewScope").PushScope("NewScope");
```

Es ist nicht möglich für ein Token eine Übergangsaktion zu definieren, wenn ihm auch eine "normale" Aktion zugewiesen ist.

Bei der Generierung von C++-Code wird die Aktion nur kopiert, d.h. es wird z.B. keine automatische Anpassung an den Zeichentyp vorgenommen.

Eine interessante Anwendung ist die Erkennung von Konstruktoren von C++-Klassen. Solche Konstruktoren werden dadurch gekennzeichnet, dass derselbe Bezeichner einmal vor und einmal nach zwei Doppelpunkten steht, z.B. CParser::CParser. Dass es sich um denselben Bezeichner

handelt kann normalerweise leicht durch semantischen Code festgestellt werden. Solcher Code wird aber bei einer Vorausschau nicht ausgeführt. Hier kann man sich dadurch behelfen dass man ein Platzhalter-Token "CLASS" definiert und ihm den Bezeichner bei seiner ersten Erkennung hinzufügt. Das zweite Vorkommen wird dann durch das CLASS-Token erkannt.

```
IsScoped  ::=
ID "::" ID


IsClass  ::=
ID.AddToken(xState.str(),   "CLASS")
"::"
CLASS


IF(IsScoped())
  IF(IsClass())
      ID.AddToken(xState.str(), "CLASS")
     "::" CLASS  {{out << "class found"; }}
  ELSE
     ID "::" ID      {{out << "member function found"; }}
  END
ELSE
  ID                 {{out << "identifier found"; }}
END
```

10.2.15 Aufrufparameter

Wird innerhalb einer Produktion eine andere Produktion aufgerufen, so müssen der letzteren die erforderlichen Parameter übergeben werden. Variablen werden in den semantischen Aktionen deklariert. Der Name einer solchen Variablen kann dann in eckige Klammern "[...]" eingefügt werden, die dem Namen einer aufgerufenen Produktion unmittelbar folgen.

Beispiel:

Die Produktion *Comment* habe den Parameter: std::str& xsComment.

Name:	Comment
Parameter:	str& xsComment
Text:	…

Eine zweite Produktion "Script" könnte die *Comment*-Produktion aufrufen. Um ihr den Parameter übergeben zu können, muss eine String-Variable zuvor deklariert werden.

Name:	Script
Parameter:	…
Text:	{{ str s;}} Comment[s] ...

10.3 Klassen-Elemente und C++-Befehle

Während des Parsens eines Eingabetextes kann zugleich eine Verarbeitung der erkannten Abschnitte erfolgen. Vor und nach jedem Erkennungsschritt können hierzu sogenannte "semantische Aktionen" ausgeführt werden, d.h. es können eine Anzahl von Programmieranweisungen ausgeführt werden.
Im Unterschied zu den meisten Programmiersprachen die zunächst in eine Form kompiliert werden müssen, die der Computer versteht, sind in TETRA die Programmieranweisungen direkt ausführbar, d.h. sie werden "interpretiert".

Der in TETRA integrierte Interpreter beherrscht

1. eine Untermenge der Befehle der Programmiersprache C++
2. einige speziell für TETRA verfasste Funktionen
3. Formatierungsanweisungen
4. einige spezielle Befehle, die dem Zugriff auf den aktuellen Zustand des Parsers dienen.
5. Benutzerdefinierte Funktionen und Variablen (Klassenelemente)

10.3.1 Eingabemaske für Klassenelemente

Die unter Punkt 5 genannten Definitionen neuer Funktionen und Variablen durch den Benutzer beruhen auf den unter Punkt 1 - 4 genannten Befehlen. In der vom TextTransformer erzeugbaren Parserklasse sind diese Funktionen und Variablen Klassenelemente.
Die Funktionen und Variablen und Funktionstabellen werden in der Liste der Klassenelemente durch unterschiedliche Symbole gekennzeichnet:


Funktion

Variable

Funktions-Tabelle

Für jedes dieser Klassenelemente gibt es die bekannten Eingabefelder.

Name:	eindeutigen Name
Typ:	C++-Variablentyp
Parameter:	C++-Parameterdeklaration (nur für Funktionen)
Kommentar:	beliebiger Kommentartext
Text/Initialisierung:	Skripttext der Funktion oder Initialisierung der Variable

Für Funktionen sind Angabe von Namen und Text erforderlich, für Variablen reicht ein Name. Erst wenn diese Felder gefüllt sind, ist es möglich das Skript in die Verwaltung zu übernehmen oder einen Kommentar zu schreiben.

10.3.1.1 Name

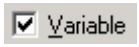
Jedem Klassenelement (Funktion oder Variable) muss ein Name gegeben werden.
Der Name kann aus den Buchstaben des Alphabets, Ziffern und Unterstrichen '_' gebildet sein, wobei der Unterstrich nicht an erster Stelle des Namens stehen darf.

Beispiele.: Fakultaet, m_Table1

Jeder neue Name muss sich in mindestens einem Zeichen von allen anderen Namen der in der Verwaltung enthaltenen Produktionen, Token, Funktionen und Variablen unterscheiden.

10.3.1.2 Typ

Je nachdem, ob in der Werkzeugleiste



das Klassenelement als Variable bestimmt ist oder nicht, ist die im Typ-Feld anzugebende Typbezeichnung der Typ der Klassenvariablen oder der Rückgabetyp einer Klassenfunktion. Im ersten Fall ist eine Typangabe zwingend erforderlich, im zweiten Fall wird der Typ "void" angenommen, falls das Feld nicht gefüllt ist.

10.3.1.3 Parameter

Je nachdem, ob in der Werkzeugleiste

das Klassenelement als Variable bestimmt ist oder nicht, ist das Parameterfeld unsichtbar oder sichtbar.
Für die Parameter einer Klassenmethode gilt im wesentlichen das gleiche, was für die Parameter der Produktionen beschrieben ist.

Impliziter xState-Parameter

Ein Problem gibt es beim Aufruf der Methode, wenn er sowohl für den Interpreter als auch für den exportierten C++-Code gültig sein soll. Im erzeugten Code der Parserklasse wird in die Deklaration der Methode automatisch als erster Parameter die xState-Variable eingefügt. Z.B.

```
void Method(state_type& xState);
```

Im Interpreter-Code eines Aufrufs der Methode kann die Übergabe des *xState*-Parameters zur Bequemlichkeit aber unterbleiben.

```
{- Method( ); -}
```

Im Interpreter wird dieser Code geparst und der fehlende Parameter wird automatisch ergänzt. Wenn

der Code aber exportiert wird, wird er schlicht kopiert, so dass der *xState*-Parameter explizit mit angegeben sein muss:

```
{_  Method(xState); _}
```

Soll der semantische Code sowohl intern als auch extern gültig sein, so sollte der Methode *xState* übergeben werden:

```
{= Method(xState); =}
```

Seit der Version 0.9.8.1 wird der xState-Parameter automatisch in den generierten Code eingefügt, wenn nicht die Nur kopieren Option gesetzt ist, und wenn es sich nicht um einen Code-Teil handelt, der nur exportierbar ist.

Durch Setzen der entsprechenden Projekt-Option kann die Erzeugung von Warnmeldungen aktiviet werden, die erscheinen, wenn ein solcher xState-Parameter fehlt.

10.3.1.4 Kommentar

Ein Kommentartext zum Klassenelement wird in dem gelblich hinterlegten Feld dargestellt.

Dieses Feld wird temporär auch zur Anzeige von Fehlemeldungen etc. verwendet. Um den Kommentar zu ändern muss der Kommentarschalter betätigt werden, der eine gesonderte Dialogbox zum Edieren des Kommentartextes öffnet.

10.3.1.5 Text/Initialisierung

Je nachdem, ob in der Werkzeugleiste

das Klassenelement als Variable bestimmt ist oder nicht, dient das Editor-Feld für Code zur einer Initialisierung der Variablen oder zur Eingabe des Funktionskörpers.

a) **Initialisierungscode**

Optional kann für jede Klassenvariable ein Initialisierungscode geschrieben werden, der jedesmal ausgeführt wird, bevor ein neuer Text geparst wird.

Beispiel:

Name: m_TotalPercent
Typ: int
Init:

```
m_TotalPercent = 100;
```

Name: m_Numerals
Typ: mstrstr
Init:
```
m_Numerals["eins"] = "one";
m_Numerals["zwei"] = "two";
m_Numerals["drei"] = "three";
m_Numerals["vier"] = "four";
m_Numerals["fünf"] = "five";
m_Numerals["sechs"] = "six";
m_Numerals["sieben"] = "seven";
m_Numerals["acht"] = "eight";
m_Numerals["neun"] = "nine";
```

b) **Funktionskörper**

Der Funktionskörper von Klassenfunktionen wird hier geschrieben.

Beispiel:

Name: abs
Typ: int
Parameter: int xi
Text:
```
if(xi < 0)
 xi = -xi;
return xi;
```

10.3.2 Liste aller Anweisungen

interpretierte C++-Anweisungen

Methoden der Klasse str

bool	**empty**() const
unsigned int	**size**() const
unsigned int	**length**() const
unsigned int	**find**(const str& xs) const
unsigned int	**find**(const str& xs, unsigned int pos) const
unsigned int	**rfind**(const str& xs) const
unsigned int	**rfind**(const str& xs, unsigned int pos) const
unsigned int	**find_first_of**(const str& xs) const
unsigned int	**find_first_of**(const str& xs, unsigned int pos) const
unsigned int	**find_first_not_of**(const str& xs) const
unsigned int	**find_first_not_of**(const str& xs, unsigned int pos) const
unsigned int	**find_last_of**(const str& xs) const
unsigned int	**find_last_of**(const str& xs, unsigned int pos) const
unsigned int	**find_last_not_of**(const str& xs) const
unsigned int	**find_last_not_of**(const str& xs, unsigned int pos) const

str	**substr**(int from, int count) const
str	**substr**(int from) const
char	operator[](int xiIndex)
void	**clear**() removes the text in the string
str&	**replace**(unsigned int pos, unsigned int len, const str& s)

Container Klassen

Methoden der Klasse vector

sämtliche Methoden der Klasse cursor und zusätzlich

cursor_type	**getCursor**() const
void	**reset**()
void	**clear**()
void	**push_back**(const value_type& xValue)
void	**pop_back**()
value_type	**back**()
value_type	**front**()
bool	**remove**()
bool	**setValue**(const value_type& xValue)
operator[](int xiIndex)	

Methoden der Klasse map

sämtliche Methoden der Klasse cursor und zusätzlich

cursor_type	**getCursor**() const
str	**key**() const
bool	**containsKey**(const str& xsKey) const
bool	**findKey**(const str& xsKey)
void	**reset**()
void	**clear**()
bool	**add**(const str& key, const value_type& xValue)
bool	**remove**()
bool	**remove**(const str& xsKey)
bool	**setValue**(const value_type& xValue)
operator[](str xsIndex)	

Methoden der Klasse cursor

bool	**isValid**() const
bool	**hasCurrent**() const
bool	**empty**() const
unsigned int	**size**() const
value_type	**value**() const
bool	**gotoNext**()
bool	**gotoPrev**()
bool	**containsValue**(const value_type& xValue) const

bool	**findValue**(const value_type& xValue)
bool	**findNextValue**(const value_type& xValue)
bool	**findPrevValue**(const value_type& xValue)

ein map-cursor hat zusätzlich die Methoden:

str	**key**() const
bool	**containsKey**(const str& xsKey) const
bool	**findKey**(const str& xsKey)

Methoden einer Funktionstabelle

bool	**add**(STRING, STRING);
void	**visit**(const node& xNode)

Methoden der Klasse d/node

d/node	**clone**()
bool	**addChildFirst**(const node& xNewChild)
bool	**addChildLast**(const node& xNewChild)
d/node	**add**(const str& xsLabel, const str& xsValue)
bool	**addChildBefore**(const node& xnNewChild, const node& xnRefChild)
d/node	**removeChild**(const node& xnOldChild)
bool	**replaceChild**(d/node& xNewChild, d/node& xOldChild)
str	**label**() const
void	**setLabel**(const str& xsLabel)
str	**value**() const
void	**setValue**(const str& xsValue)
unsigned int	**id**() const
void	**setId**(unsigned int xuiId)
bool	**hasChildren**() const
bool	**isDescendant**(const node& xNode) const
bool	**isAncestor**(const node& xNode) const
bool	**isSibling**(const node& xNode) const
unsigned int	**level**() const
unsigned int	**descendentsCount**() const
unsigned int	**childCount**() const
d/node	**root**() const
d/node	**parent**() const
d/node	**firstChild**() const
d/node	**lastChild**() const
d/node	**nextSibling**() const
d/node	**prevSibling**() const
d/node	**firstSibling**() const
d/node	**lastSibling**() const
d/node	**bottomFirstChild**() const
d/node	**bottomLastChild**() const
d/node	**next**() const

d/node	**follow**() const
d/node	**prev**() const
d/node	**nextLeaf**() const
d/node	**prevLeaf**() const
d/node	**findNextLabel**(const str& xsLabel) const
d/node	**findNextLabel**(const str& xsLabel, const node& xnLast) const
d/node	**findNextValue**(const str& xsValue) const
d/node	**findNextValue**(const str& xsValue, const node& xnLast) const
d/node	**findNextId**(unsigned int xuiId) const
d/node	**findNextId**(unsigned int xuiId, const node& xnLast) const
d/node	**findPrevLabel**(const str& xsLabel) const
d/node	**findPrevLabel**(const str& xsLabel, const node& xnLast) const
d/node	**findPrevValue**(const str& xsValue) const
d/node	**findPrevId**(unsigned int xuiId) const
d/node	**findPrevId**(unsigned int xuiId, const node& xnLast) const
d/node	**findPrevValue**(const str& xsValue, const node& xnLast) const
d/node	**findChildLabel**(const str& xsLabel, bool xbRecursive = true)
d/node	**findChildValue**(const str& xsValue, bool xbRecursive = true)
d/node	**findChildId**(unsigned int xuiId, bool xbRecursive = true)
d/node	**findParentLabel**(const str& xsLabel)
d/node	**findParentValue**(const str& xsValue)
d/node	**findParentId**(unsigned int xuiId)
void	**setAttrib**(const str& xsLabel, const str& xsValue)
str	**attrib**(const str& xsLabel)
bool	**hasAttrib**() const
void	**sortCildrentA**()
void	**sortCildrenD**()

Funktionen zur String-Manipulation

stod	Konvertierung eines str nach double
stoi	Konvertierung eines str nach int
hstoi	Konvertierung eines hexadezimalen Strings nach int
stoc	Konvertierung eines Strings nach char
dtos	Konvertierung eines double-Werts in einen str
itos	Konvertierung eines int-Werts in einen str
itohs	Konvertierung eines int-Werts in einen str
ctohs	Konvertierung eines Zeichens in einen hexadezimalen str
ctos	Konvertierung eines Zeichens in einen str

to_upper_copy	Gibt einen str in Großschreibung zurück
to_lower_copy	Gibt einen str in Kleinschreibung zurück
trim_left_copy	entfernt führende Leerzeichen
trim_right_copy	entfernt abschließende leerzeichen
trim_copy	entfernt Leerzeichen an den Enden eines str

Dateibehandlung

basename	Liefert den Basisnamen eines Dateinamens
extension	Liefert die Erweiterung eines Dateinamens
change_extension	Ändert die Erweiterung eines Dateinamens
append_path	Setzt Pfade zusammen
current_path	Liefert den aktuellen Pfad
exists	Prüft die Existenz eines Pfades
is_directory	Prüft, ob der Pfad ein Verzeichnis bezeichnet
file_size	Liefert die Dateigröße
find_file	Sucht eine Datei in einem Verzeichnis
load_file	Lädt den Inhalt einer Datei
path_separator	String-Konstante für den Pfad-Trenner

Ausgabe

out	Schreiben in die Ausgabe
log	Schreiben von Log-Informationen
bool_bin	schreibt *bool* binär
int_bin	schreibt *int* binär
uint_bin	schreibt *unsigned int* binär
float_bin	schreibt *float* binär
double_bin	schreibt *double* binär
char_bin	schreibt *char* binär
string_bin	schreibt *char** der Länge des strings.
bin	schreibt übergebenen Typ binär
ends	schreibt binäres Null-Zeichen

Formatierungsanweisungen

unsigned int	**size**() const
str	**str**() const
void	**parse**(const str& xs)

Sonstige Funktionen

clock_sec	Zeitberechnung
time_stamp	Datums- und Zeit-String
time_stamp(const str& xsFormat)	
random	Generiert Zufallszahlen
throw	Wurf einer Ausnahmeklasse
d/node	detach_node(const d/node& xn)

Parserklasse-Methoden

Parserzustand

unsigned int	**size**() const
unsigned int	**length**(int sub = 0) const
bool	**matched**(int sub) const
bool	**matched**() conststr **str**(int sub) const
str	**str**() const
str	**text**(unsigned int from) const
str	**text**(unsigned int from, unsigned int to) const
str	**copy**() const
int	**LastSym**() const
unsigned int	**Line**() const
int	**Col**() const
unsigned int	**Position**() const
unsigned int	**LastPosition**() const
unsigned int	**NextPosition**() const
void	**SetPosition**(unsigned int xi);
bool	**IsSubCall**() const
str	**ProductionName**() const
str	**BranchName**() const
str	**next_str**() const
str	**next_copy**() const
stri	**next_str**(int sub) const
unsigned int	**next_size**() const
unsigned int	**next_length**(int sub = 0) const
str	**lp_str**() const
str	**lp_str**(int sub) const
str	**lp_copy**() const
unsigned int	**lp_length**(int sub = 0) const
str	**la_str**() const
str	**la_copy**() const
str	**la_str**(int sub) const
unsigned int	**la_length**(int sub = 0) const
int	**GetState**()
void	**SetState**(int xeState);

Plugin-Methoden

str	**SourceName**()
void	**SourceName**(const str& xsSourceName, bool xbLast)
str	**TargetName**()
void	**TargetName**(const str& xsTargetName)
str	**SourceRoot**()
void	**SourceRoot**(const str& xsSourceDir)
str	**TargetRoot**()
void	**TargetRoot**(const str& xsTargetDir)

```
void        RedirectOutput( const str& xsFilename )
void        RedirectOutputBinary( const str& xsFilename )
void        RedirectOutput( const str& xsFilename, bool xbAppend )
void        RedirectOutputBinary( const str& xsFilename, bool xbAppend )
void        ResetOutput( )

dnode       GetDocumentElement();
void        WriteDocument();
void        WriteDocument(const str& xsFilename);

indent
str         IndentStr() const
void        SetIndenter(char xc)
void        PushIndent( int xi )
void        IncrIndent( int xi )
void        PopIndent()
void        ClearIndents()

void        PushScope(const str& xs)
void        PopScope()
void        ClearScopes()
str         ScopeStr() const

bool        AddToken( const str& xsText,
                      const str& xsDynTokenName)
bool        AddToken( const str& xsText,
                      const str& xsDynTokenName,
                      const str& xsScope)

void        UseExcept(bool xbUseExcept)
bool        GetUseExcept() const
bool        HasError() const
void        GenError(const str& xs)
void        AddMessage(const str& xs)
void        AddWarning(const str& xs)
void        AddError(const str& xs)
```

10.3.3 interpretierte C++-Anweisungen

Die Syntax des Interpreters ist einfache C++-Syntax. Sämtlicher Code, der im Interpreter funktioniert, ist als exportierter Code auch mit externen C++-Compilern compilierbar und ausführbar. Umgekehrt gilt dies nicht. Trickreiche Formulierungen oder Code ohne Auswirkungen (z.B. Rückgabe eines Werts ohne ihn zuzuweisen) können Fehler mit unverständlichen Fehlermeldungen im Interpreter erzeugen.)

Tip.: Schreiben sie einfachen Code und verwenden sie besser zwei Anweisungen, als eine.

10.3.3.1 C++

Für den Programmierneuling seien hier ein paar allgemeine Bemerkungen vorausgeschickt.

C++ ist eine weit verbreitete sehr komplexe Programmiersprache. Eine für die Bearbeitung von Texten sinnvolle Untermenge der Befehle dieser Sprache, die auch vom Programmierlaien leicht zu beherrschen ist, ist in TETRA integriert

Generell besteht eine Programmiersprache in einer Reihe von **Anweisungen** oder Befehlen, die nacheinander ausgeführt werden. Programmieranweisungen nehmen zumeist bestimmte Operationen an Daten vor. Die Daten wiederum werden in Programmiersprachen durch Variablen repräsentiert. Z.B. könne eine Variable mit dem Namen T den Text "tetra" enthalten und die Anweisung *to_upper_copy*(T) würde diesen Text in die Großschreibung "TETRA" transformieren.

Bei C++ handelt es sich um eine sogenannte **typisierte** Sprache, d.h. verschiedene Arten von Inhalten (Daten) werden durch verschiedene Typen von Variablen repräsentiert, die nicht ohne weiteres ineinander umgewandelt werden können. So sind z.B. Texte keine Zahlen. Der Versuch einer Umwandlung von des Textes "TETRA" in eine Zahl wäre ein sinnloses Unterfangen. In einer nicht typisierten Programmiersprache, wäre eine solche Umwandlung erlaubt und hätte schwer vorhersehbare Folgen. Typisierte Programmiersprachen sind also sicherer als andere, d.h. die Gefahr Fehler zu machen ist geringer. Andererseits erfordern typisierte Sprachen einen Mehraufwand vom Programmierer. Dieser muss Variable bevor er sie benutzen kann zunächst mit ihrem Typ **deklarieren**.

10.3.3.2 Variablentypen

Im TETRA-Interpreter gibt elementare Variablentypen, Strings, Knoten (node) und Container-Typen und Cursor-Typen:

Typ	Wertebereich	Default-Wert (*)
bool	true/false (bzw. 1/0)	false
char	0-255	'\0'
int	-32768 - +32767	0
unsigned int	0 - +65565	0
double	-1,7E+308 - +1,7E+308 (15 Stellen)	0.0
str		""
node		node::npos
vector		leer
map		leer
cursor		--
Funktionstabelle		leer

* Default-Werte haben nur für den Interpreter Gültigkeit. Für den exportierten Code müssen sie bei Bedarf explizit gesetzt werden.

10.3.3.2.1 bool

Syntax

bool <bezeichner>;

Beschreibung

Das Schlüsselwort **bool** kennzeichnet einen Datentyp, der nur die Werte **false** oder **true** aufnehmen kann. Die Schlüsselwörter **false** und **true** sind Boolesche Konstanten mit vordefinierten Werten. Das numerische Äquivalent für **false** ist Null und **true** entspricht der Eins.

Ein Wert vom Typ bool lässt sich in einen Wert vom Typ **int** konvertieren. Die numerische Umwandlung setzt **false** auf den Wert Null und **true** auf den Wert Eins.

Umgekehrt ist es möglich **double** oder **int** Werte in Werte vom Typ **bool** zu konvertieren. Dabei wird eine arithmetische Null in den Wert **false**, und jeder andere Wert zu **true** umgewandelt.

10.3.3.2.2 char

Syntax

char<variablen-name>

Beschreibung

Dieser Typ-Bezeichner dient zur Deklaration einer Zeichenvariablen; Variablen vom Typ char belegen 1 Byte an Speicherplatz.

Objekte, die als Typ char deklariert sind, können jedes Zeichen aus dem nicht erweiterten ASCII-Zeichensatz aufnehmen. Ein auf dem Bildschirm nicht darstellbares Zeichen kann durch eine Escape-Sequenz repräsentiert werden.

10.3.3.2.3 int

Syntax

int <bezeichner> ;

Beschreibung

Das Typ-Bezeichner **int** kennzeichnet einen Datentyp, der die numerischen Werte aus dem Bereich -32768 - +32767 annehmen kann.

Integer-Konstanten können auch in oktaler oder hexadezimaler Schreibweise angegeben werden.

10.3.3.2.4 unsigned int

Syntax

unsigned int <bezeichner> ;

Beschreibung

Das Typ-Bezeichner **unsigned int** kennzeichnet einen Datentyp, der die numerischen Werte aus dem Bereich 0 - +65565 annehmen kann.
Durch den Typmodifizierer **unsigned** wird angezeigt, dass der Wert der Variablen stets positiv ist

Integer-Konstanten können auch in oktaler oder hexadezimaler Schreibweise angegeben werden.

10.3.3.2.5 double

Syntax

double <bezeichner>

Beschreibung

Der Typ-Bezeichner **double** wird bei der Deklaration von Namen verwendet, die für Gleitkommatypen stehen.

10.3.3.3 str

Syntax

str <bezeichner> ;

Beschreibung

str ist die Interpreter-Version von std::string oder std::wstring. Im erzeugten Code ist **str** definiert durch: *typedef std::string str*, bzw. für Unicode-Parser durch: *typedef std::wstring str*.

Eine *str*-Variable enthält eine Kette von Zeichen, d.h. einen Text. Zu diesen Zeichen zählen auch die Steuerzeichen, die aus der Kombination eines Backslashs mit anderen Zeichen bestehen. Z.B. '\n' ist das Zeilenumbruchszeichen. Um ein ein Backslash-Zeichen in seiner literalen Bedeutung in einem String zu verwenden, muss ihm ein zweites Backslash-Zeichen vorangestellt werden: '\\'

Beispiel:

```
str  s  =  "C:\\TextTransformer\\bin";
// falsch:   "C:\TextTransformer\bin"
```

Weitere auf dem Bildschirm nicht darstellbares Zeichen können durch andere Escape-Sequenzen repräsentiert werden.

Aneinander angrenzende String-Literale werden zu einem einzigen String-Literal verschmolzen. So

kann man längere Texte übersichtlicher gestalten.

Beispiel:

```
str  s  =  "/***************************\n"
          "*   Dies ist ein Kommentar    *\n"
           "***************************/\n";
```

Im Unterschied zu den bisherigen Variablentypen ist *str* kein Grundtyp sondern ein abgeleiteter Typ (eine Klasse), der über Methoden verfügt, die Informationen über den enthaltenen Text liefern oder dessen Manipulation erlauben:

a) **Befehle, die Textinformationen liefern**

bool **empty**() const	gibt zurück, ob der String leer ist
unsigned int **size**() const	gibt die Anzahl der Zeichen des Strings zurück
unsigned int **length**() const	gibt die Anzahl der Zeichen des Strings zurück

Eine ganze Familie von Methoden dient dem Auffinden bestimmter Positionen innerhalb eines Strings.

Teilstrings:

void **substr**(int from, int count) const
void **substr**(int from) const

gibt den Abschnitt des Strings zurück, der mit dem Zeichen beginnt, dessen Index *from* ist und der die Länge *count* hat. Falls der String keine *count* Zeichen mehr enthält, werden so viele zurückgegeben wie enthalten sind. Wird der zweite Parameter weggelassen, so werden alle Zeichen ab from zurückgegeben.

Index-Operator

Einzelne Zeichen des Strings können über ihren Index ermittelt werden. Das erste Zeichen hat den Index 0 und das letzte den Index size() - 1.

```
str s = "TextTransformer";
char c = s[5];  // jetzt gilt: c == 'r'
```

Es ist nicht möglich einzelne Zeichen über den Index des Strings zu ändern. Verwenden sie hierzu die **replace**-Methode (s.u.)

b) **Manipulation von Texten**:

void **clear**() löscht alle Zeichen aus dem String

str& **replace**(unsigned int pos, unsigned int len, const str& s)

Ersetzt (maximal) len Zeichen ab der Position pos durch sämtliche Zeichen von s.
Eine Referenz auf den String selbst wird zurückgegeben.

Beispiel:

```
str s = "wood";
out << s.replace(0, 1, "g") << " ";
out << s;
```

Ausgabe: good good

10.3.3.3.1 Suche

Eine Reihe von Methoden dient dem Auffinden bestimmter Positionen innerhalb eines Strings *str*.

unsigned int **find**(const str& xs) const
unsigned int **find**(const str& xs, unsigned int pos) const
unsigned int **rfind**(const str& xs) const
unsigned int **rfind**(const str& xs, unsigned int pos) const
unsigned int **find_first_of**(const str& xs) const
unsigned int **find_first_of**(const str& xs, unsigned int pos) const
unsigned int **find_first_not_of**(const str& xs) const
unsigned int **find_first_not_of**(const str& xs, unsigned int pos) const
unsigned int **find_last_of**(const str& xs) const
unsigned int **find_last_of**(const str& xs, unsigned int pos) const
unsigned int **find_last_not_of**(const str& xs) const
unsigned int **find_last_not_of**(const str& xs, unsigned int pos) const

unsigned int **find**(const str& xs) const
unsigned int **find**(const str& xs, unsigned int pos) const

Gibt den Index des ersten Vorkommens von xs im String zurück, fall xs vorkommt. Andernfalls wird eine spezielle Konstante *str::pos zurückgegeben.* (*str::npos* kann nur im Kontext von Gleichheitsopratoren verwendet werden.)

```
str s = "hello world";
unsigned int pos = s.find("o world");
if(pos != str::npos)
  out << s.substr(0, pos);
// else Wert von pos nicht benutzen
```

Ausgabe: hell

Der **find**-Funktion kann als zweiter Parameter die Position übergeben werden, von der ab gesucht werden soll.

```
str s = "C:\\Programme\\TextTransformer\\Target\\test.txt";
unsigned int pos = s.find("\\");
unsigned int lastpos = str::npos;
```

```
while(pos != str::npos)
{
   lastpos = pos + 1;
   pos = s.find("\\", lastpos);
}
if(lastpos != str::npos)
  out << s.substr(0, lastpos);   // ergibt : C:\Programme\TextTransformer\Target\
```

unsigned int **rfind**(const str& xs) const
unsigned int **rfind**(const str& xs, unsigned int pos) const

Die beiden *rfind*-Methoden funktionieren analog zu den entsprechenden *find*-Methoden, jedoch wird der String rückwärts durchsucht.
Das Ergebnis des letzten Beispieles wird mit der *rfind*-Methode schneller gefunden:

```
str  s  =  "C:\\Programme\\TextTransformer\\Target\\test.txt";
unsigned int pos = s.rfind("\\");
if(pos != str::npos)
  out << s.substr(0, pos + 1);   // ergibt : C:\Programme\TextTransformer\Target\
```

unsigned int **find_first_of**(const str& xs) const
unsigned int **find_first_of**(const str& xs, unsigned int pos) const
unsigned int **find_first_not_of**(const str& xs) const
unsigned int **find_first_not_of**(const str& xs, unsigned int pos) const
unsigned int **find_last_of**(const str& xs) const
unsigned int **find_last_of**(const str& xs, unsigned int pos) const
unsigned int **find_last_not_of**(const str& xs) const
unsigned int **find_last_not_of**(const str& xs, unsigned int pos) const

Mittels dieser Funktionen wird nicht nach einem Teil-String gesucht, sondern nach dem Vorkommen eines der Zeichen aus dem String-Argument *xs*. Mit dem optionalen zweiten Argument *pos* kann eine Position besimmt werden, ab der der String durchsucht werden soll.

find_first_of
sucht vorwärts nach dem ersten Zeichen aus *xs*

find_first_not_of
sucht vorwärts nach dem ersten Zeichen das nicht in *xs* enthalten ist

find_last_of
sucht rückwärts nach dem ersten Zeichen aus *xs*

find_last_not_of
sucht rüchwärts nach dem ersten Zeichen das nicht in *xs* enthalten ist

Beispiel:

Der folgende Code fügt Backslashes vor einzelnen Backslahes, vor dem Zeilenumbruchszeichen und vor den Wagenrücklaufzeichen ein. Mit xs = "Hello\r\nGood bye", erhält man das Ergebnis: "\"Hello\\r\\nGood bye\"".

```
str sResult = "\"";
str sFindWhat = "\\\r\n";
unsigned int oldpos = 0;
unsigned int pos = xs.find_first_of(sFindWhat);
while(pos != str::npos)
{
   sResult += xs.substr(oldpos, pos - oldpos);
  switch(xs[pos])
  {
    case '\\':
     sResult += "\\";
    break;
    case '\r':
     sResult += "\\r";
    break;
    case '\n':
     sResult += "\\n";
    break;
  }
  oldpos = pos + 1;
   pos = xs.find_first_of(sFindWhat, oldpos);
}

sResult += xs.substr(oldpos);
sResult += "\"";
return sResult;
```

10.3.3.4 Container

Container enthalten und verwalten Ansammlungen von elementaren Variablen.
TETRA kennt zwei Grundtypen von Containern und einen abgeleiteten Typ:

vector
map
stack

Über die Elemente eines Containers kann iteriert werden oder es kann nach speziellen Elementen gesucht werden. hierzu dient ein

cursor

Jeder Container enthält intern einen Cursor und die Methoden des Cursors werden direkt als Methoden des Containers aufgerufen. Ein Cursor kann aber auch extern deklariert werden.

Die Namen der Container- und Cursor-Typen werden aus einer Bezeichnung für die Art des Containers und der Kurzbezeichnung der enthaltenen Variablen-Typen gebildet.

Typ	**vector**	**map**	**Fkt-Tabelle**
bool	vbool	mstrbool	bool_mstrfun
int	vint	mstrint	int_mstrfun
unsigned int	vuint	mstruint	uint_mstrfun
double	vdbl	mstrdbl	dbl_mstrfun
char	vchar	mstrchar	char_mstrfun

str	vstr	mstrstr	str_mstrfun
node	vnode	mstrnode	node_mstrfun

Eine sehr spezielle Art von Container zur Auswertung von Parse-Bäumen ist eine

Funktionstabelle.

Für Funktionstabellen gibt es keinen Cursor.

10.3.3.4.1 Vector

In der Containerklasse Vector wird eine Liste von Elementen verwaltet. Alle Elemente einer Liste haben den gleichen Datentyp. Es gibt jedoch verschiedene Vector-Typen, je nach Typ der enthaltenen Elemente. (Siehe obige Tabelle)

Syntax

vbool <bezeichner>
vint <bezeichner>
vuint <bezeichner>
vchar <bezeichner>
vstr <bezeichner>
vnode <bezeichner>

Beschreibung

Die Reihe der Vector-Typen sind im exportierten Code ***typedefs*** der Klasse CTT_Vector< value_type >. CTT_Vector ist abgeleitet von std::vector< value_type >

Sämtliche Vector-Klassen verfügen über die gleichen Methoden. Dies sind

1. die nur lesenden Methoden der allgemeinen Cursor-Klasse

bool	**isValid**() const
bool	**hasCurrent**() const
bool	**empty**() const
unsigned int	**size**() const
bool	**gotoNext**()
bool	**gotoPrev**()
value_type	**value**() const
bool	**containsValue**(const value_type&) const
bool	**findValue**(const value_type& xValue)
bool	**findNextValue**(const value_type& xValue)
bool	**findPrevValue**(const value_type& xValue)

2.) **getCursor**() const

Mit dieser Methode wird die Verbindung eines externen Cursors mit dem Vector hergestellt.

Beispiel:

```
vint v;
vint::cursor cr =  v.getCursor();
```

cr zeigt nun auf das gleiche Datenelement wie der interne Cursor, kann aber unabhängig vom ersten neu positioniert werden.

3) Methoden, die den Inhalt des Vectors verändern

void **reset**()

Setzt den Cursor vor bzw. hinter die Liste der Elements des Vectors.

void **clear**()

entfernt alle Elemente und führt *reset* aller Cursor durch.

void **push_back**(const value_type& xValue)

fügt xValue als neues Element am Ende des Vectors an. Für alle Cursor wird *reset* durchgeführt.

void **pop_back**()

entfernt das letzte Element aus dem Vector. Für alle Cursor wird *reset* durchgeführt. Wenn der Vector leer ist, wird ein Fehler erzeugt. Zur Sicherheit sollte daher vor dem Aufruf von pop*_back* zunächst geprüft werden, ob der Vector Elemente enthält:

```
if(v.size())
  v.pop_back();
```

bool **remove**()

löscht das aktuelle Element aus dem Vector und gibt bei Erfolg *true* zurück. Gibt es kein aktuelles Element oder tritt ein Fehler auf, wird *false* zurückgegeben. Für alle Cursor wird *reset* durchgeführt.

bool **setValue**(const value_type& xValue)

Setzt den Wert des aktuellen Elements als *xValue* und gibt bei Erfolg *true* zurück. Gibe es kein aktuelles Element oder tritt ein Fehler auf, so gibt die Funktione *false* zurück.

4) Direkter Zugriff auf die Elemente

value_type **back**()

liefert das letzte Element zurück. Wenn der Vector leer ist, wird ein Fehler erzeugt. Zur Sicherheit sollte daher vor dem Aufruf von *back* zunächst geprüft werden, ob der Vector Elemente enthält:

```
if(v.size())
  value = v.back();
```

value_type **front**()

liefert das erste Element zurück. Wenn der Vector leer ist, wird ein Fehler erzeugt. Zur Sicherheit sollte daher vor dem Aufruf von *front* zunächst geprüft werden, ob der Vector Elemente enthält:

```
if(v.size())
  value = v.front();
```

Index-operator

Auf die Elemente, die zuvor mit der **push_back** Methode in den Vektor eingefügt wurden, kann mittels ihres indexes direkt zugegriffen werden. Das erste Element hat den Index 0 und das letzte den Index size() -1. Wenn ein vstr namens *v* existiert, so kopiert

```
str s = v[0];
```

das erste Element in den String *s* und

```
v[0] = s;
```

kopiert den str *s* in das erste Element, falls es existiert. Wenn nicht zuvor ein Element mit der **push_back** Methode in den Vektor eingefügt wurde, erscheint eine Fehlermeldung.

10.3.3.4.1.1 Stack

Es gibt keine eigene allgemeine Stack-Container-Klasse im TextTransformer-Interpreter. Dennoch lassen sich Stacks leicht realisieren.

Zunächst sei auf die beiden speziellen Stacks zur Verwaltung von Textbereichen und von Einrückungen hingewiesen. Auch ergibt sich ein Stack automatisch aus rekursiv aufgerufenen Produktionen (s.u.).

In anderen Fällen kann ein Vector als Stack verwendet werden. Mit *push_back* kann ein neuer Wert auf den Stack gelegt werden, mit *back* wird er gelesen und mit *pop_back* kann er wieder entfernt werden.

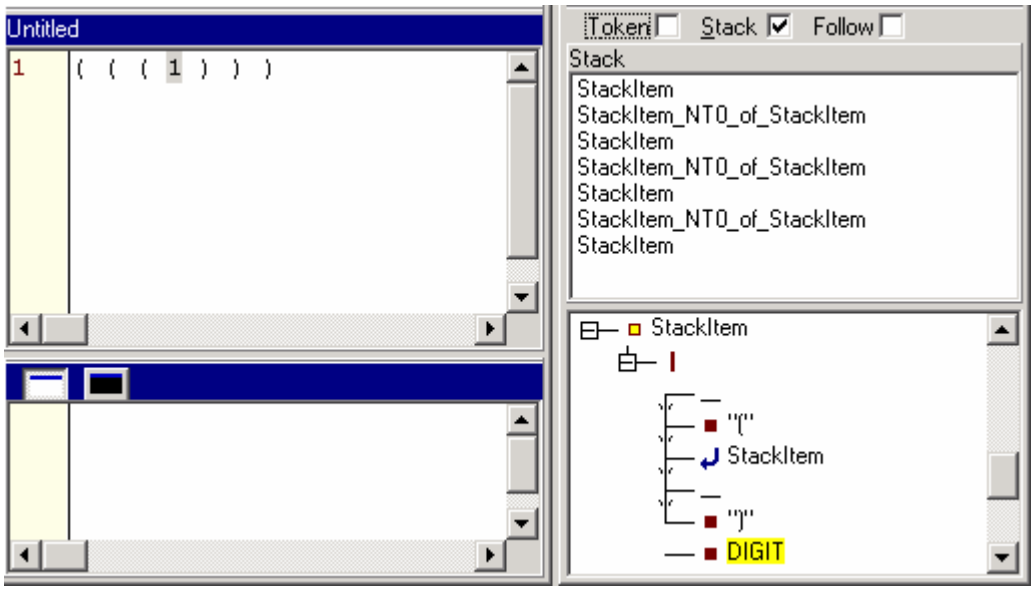
```
vint v;

for(int i = 1; i <= 3; i++)
   v.push_back(i);

while(v.size())
{
```

```
  out << v.back();
  v.pop_back();
}

// ergibt: 321
```

Automatisch ergibt sich ein Stack z.B. beim Parsen des Textes: "(((1)))" mit folgender Startregel:

```
StackItem(int xi)

{{ int i = xi + 1; }}
"("
StackItem[i]
{{ out << i << endl; }}
")"
| DIGIT
```

Für jede neue Stack-Instanz wird in einer lokalen Variablen i ein neuer Wert "auf den Stack gelegt", der beim Verlassen der Stack-Produktion wieder "vom Stack genommen" wird.



10.3.3.4.2 Map

In der Containerklasse Map wird eine Liste von Wertepaaren verwaltet. Alle Wertepaare einer Liste haben bestehen aus den gleichen Datentypen. Es gibt jedoch verschiedene Map-Typen, je nach Typ der enthaltenen Wertepaare. (Siehe obige Tabelle)

Syntax

mstrbool <bezeichner>
mstrint <bezeichner>
mstruint <bezeichner>
mstrchar <bezeichner>

mstrstr <bezeichner>
mstrnode <bezeichner>
mstrdnode <bezeichner>

Beschreibung

Die Reihe der Map-Typen sind im exportierten Code ***typedefs*** der Klasse CTT_Map< value_type >. CTT_Map ist abgeleitet von std::map< value_type >

Eine **Map**-Variable ist eine Liste von Wertepaaren.. Jedes diese Wertepaare besteht aus einem Schlüsselwort und dem dazugehörigen Wert. Die Wertepaare sind stets alphabetisch gemäß den Schlüsseln sortiert. Daher kann in einer Map sehr effektiv nach einem Schlüssel gesucht werden, während die Suche nach einem Wert vergleichsweise langsam ist.
Eine Map befindet sich jederzeit in einem bestimmten Zustand: ein bestimmtes Wertepaar kann als das aktuelle ausgezeichnet sein. Das aktuelle Wertepaar wird durch den **Cursor** (Zeiger) bezeichnet. Der Cursor kann auch ungültig sein oder bildlich gesprochen vor bzw. hinter sämtlichen Wertepaaren positioniert sein. In diesem Fall ist die Eigenschaft **hasCurrent** *false* und ein Zugriff aus den Schlüssel oder den Wert macht keinen Sinn. Der Cursor läßt sich positionieren und die Eigenschaften des aktuellen Wertepaars lassen sich abfragen oder verändern.

Sämtliche Map-Klassen verfügen über die gleichen Methoden. Dies sind

1. die nur lesenden Methoden der allgemeinen Cursor-Klasse

bool	**isValid**() const
bool	**hasCurrent**() const
bool	**empty**() const
unsigned int	**size**() const
bool	**gotoNext**()
bool	**gotoPrev**()
value_type	**value**() const
bool	**containsValue**(const value_type&) const
bool	**findValue**(const value_type& xValue)
bool	**findNextValue**(const value_type& xValue)
bool	**findPrevValue**(const value_type& xValue)

2.) **getCursor**() const

Mit dieser Methode wird die Verbindung eines externen Cursors mit der Map hergestellt.

Beispiel:

```
mstrnode m;
mstrnode::cursor cr =  m.getCursor();
```

cr zeigt nun auf das gleiche Datenelement wie der interne Cursor, kann aber unabhängig vom ersten neu positioniert werden.

3. Zusätzlich zu den allgemeinen Methoden verfügt sowohl die Map selbst als auch ein externer

Map-Cursor über die lesenden Methoden:

str **key**() const

gibt den Schlüssel des aktuellen Wertepaars zurück, oder einen leeren String, falls es kein aktuelles Wertepaar gibt.

bool **containsKey**(const str& xsKey) const

gibt *true* zurück, wenn der Schlüssel x*sKey* in der Map enthalten ist.

bool **findKey**(const str& xsKey)

sucht nach dem Schlüssel *xsKey*. Ist der Schlüssel vorhanden, wird das entsprechende Wertepaar zum aktuellen Wertepaar und die Funktion gibt *true* zurück. Andernfalls wird false zurückgegeben.

4.) Methoden, die den Inhalt der Map verändern

void **reset**()

Setzt den Cursor vor bzw. hinter die Liste der Wertepaare der Map.

void **clear**()

entfernt alle Wertepaare und führt *reset* für alle Cursor durch.

bool **add**(const str& key, const value_type& xValue)

fügt ein neues Wertepaar unter dem Schlüssel *key* mit dem Wert xValue ein und gibt bei Erfolg *true* zurück. Falls bereits ein Wertepaar mit dem Schlüssel *key* in der Map vorhanden war, wird *false* zurückgegeben und der vohandene Wert wird nicht überschrieben.
Für alle Cursor wird *reset* ausgeführt.

bool **remove**()

löscht das aktuelle Wertepaar aus der Map und gibt bei Erfolg *true* zurück. Gibt es kein aktuelles Wertepaar oder tritt ein Fehler auf, wird *false* zurückgegeben.
Für alle Cursor wird *reset* ausgeführt.

bool **remove**(const str& xsKey)

löscht das Wertepaar mit dem Schlüssel xsKey aus der Map und gibt bei Erfolg *true* zurück. Ist der Schlüssel nicht in der Map vorhanden, wird *false* zurückgegeben.
Für alle Cursor wird *reset* ausgeführt.

bool **setValue**(const value_type& xValue)

Setzt den Wert des aktuellen Wertepaars als *xValue* und gibt bei Erfolg *true* zurück. Gibe es kein aktuelles Wertepaar oder tritt ein Fehler auf, so gibt die Funktione *false* zurück.

5.) Direkter Zugriff auf Elemente

Bei einer Map m kann mit

```
m[key]
```

direkt auf den Wert zugegriffen werden und oder es können direkt Werte mit dem Schlüssel *key* in die Map eingefügt werden, wenn der Schlüssel noch nicht in der Map vorhanden war. Mit

```
m["one"] = "two";
```

wird ein Element mit dem Schlüssel "one" und dem Wert "two" in eine mstrstr eingefügt und mit

```
str s = m["one"];
```

kann der Wert "two" dann in den str *s* kopiert werden.
Wurde ein neues Element eingefügt, so wird für alle Cursor *reset* ausgeführt.

Vorsicht: wenn nicht zuvor ein Wert zu dem Schlüssel gesetzt wurde, wird durch die letzte Anweisung automatisch ein leerer String als Wert zum Schlüssel "one" eingesetzt. Der leere String wird dann nach *s* kopiert. Um zu testen, ob ein Schlüssel in der map vorhanden ist, muss die obige **containsKey** Funktion verwendet werden.

10.3.3.4.3 cursor

Zu jedem Container-Typ gehört ein entsprechender Cursor-Typ. Mittels des Cursors können einzelne Elemente des Containers gesucht werden und der Inhalt des Elements kann gelesen werden. Jeder Container enthält intern einen Cursor und die Methoden des Cursors werden direkt als Methoden des Containers aufgerufen. Ein Cursor kann aber auch extern deklariert werden. Die **Typ-Bezeichnung des Cursors** ergibt sich dabei aus der Bezeichnung des zugehörigen Containers, der der Ausdruck: "::cursor" angehängt wird.

Beispiel:

Benutzung des internen Cursors:

```
vstr v;
v.push_back("tt");
v.push_back("TETRA");
v.reset();
while ( v.gotoNext() )
  out << v.value() << endl;
```

oder Deklaration eines externen Cursors:

```
vstr v;
```

```
v.push_back("tt");
v.push_back("TETRA");
vstr::cursor  cr  =  v.getCursor();
while ( cr.gotoNext() )
  out << cr.value() << endl;
```

Der direkte Aufruf des internen Cursors wird wegen seiner Einfachheit der Deklaration eines externen Cursors meist vorgezogen werden. Soll aber beispielsweise ein Bereich des Containers markiert werden, so ist ein externer Cursor notwendig.
Auch hat die Verwendung des internen Cursors einen Nebeneffekt. Durch die Veränderung der Position des internen Cursors wird der Zustand des Containers verändert, da die Position seines Cursors Teil diese Zustandes ist. In einem const-Parser kann der interne Cursor daher nicht verwendet werden.

Der Cursor selbst kann **drei Zustände** haben:

1. er steht vor dem ersten bzw. hinter dem letzten Element: die Funktion *hasCurrent* ergibt dann *false*.

cr.hasCurrent() == false

2. er bezeichnet einen Wert. Nun gilt

cr.hasCurrent() == true

Sobald der Inhalt des Containers durch Hinzufügen oder Entfernen eines Elements verändert wird, wird der Cursor-Position (nicht der Cursor selbst) wieder ungültig, d.h es gilt wieder:

cr.hasCurrent() == false

3. er kann ungültig sein, wenn keine Verbindung zu einem Container besteht.

cr.isValid() == false

Eine Verbindung zu einem Container wird hergestellt durch den Aufruf der Container-Methode *getCursor* (s.Beispiel oben). Sie kann aber verlorengehen, wenn der Container aufhört zu existieren.

Beispiel:

```
vstr::cursor GetInvalidCursor( )
{
  vstr v;
  return vstr::cursor(v);
}
```

Methodenaufrufe eines ungültigen Cussors erzeugen keinen Fehler, sind aber nutzlos.
Im Unterschied zu einem externen Cursor ist ein interner Cursor ist niemals ungültig; seine Existenz endet mit der des Containers.

10.3.3.4.3.1 Allgemeine Cursor-Methoden

Ein Cusor erlaubt nur lesende Zugriffe auf den zugeordneten Container. Die Cursors verschiedener Arten von Containern (Vector und Map) haben eine Menge von Methoden gemeinsam, die im folgenden augelistet und erklärt werden. Als **Element** wird dabei entweder ein einzelner Wert bezeichnet, der in einem Vector enthalten ist, oder ein Wertepaar einer Map.

bool **isValid**() const

gibt zurück, ob der Cursor mit einem Container verbunden ist. Für einen internen Cursor ist dies immer der Fall.

bool **hasCurrent**() const

gibt *true* zurück, wenn der Cursor auf ein aktuelles Element zeigt. Andernfalls wird *false* zurückgegeben.

bool **empty**() const

gibt zurück, ob der Container keine Elemente enthält. Falls der Cursor ungültig ist, wird *true* zurückgegeben.

unsigned int **size**() const

gibt die Anzahl der Elemente zurück. Falls der Cursor ungültig ist, wird *0* zurückgegeben.

value_type **value**() const

gibt den Wert des aktuellen Elements zurück. Falls es kein aktuelles Element gibt, wird der Default-Wert des Elementtyps zurückgegeben: ein leerer String für den String-Typ, 0 für numerische Typen und *false* für deb bool-Typ.

Falls der Wert des Elements vom Typ *node* ist, kann mit value().label() oder value().value() das Label bzw. der Wert des Knotens erhalten werden..

bool **gotoNext**()

Setzt den Cursor auf das nächste Element, das somit zum aktuellen wird. Stand der Cursor bereits auf dem letzten Element, so gibt die Funktion den Wert false zurück und positioniert den Cursor vor das erste bzw. hinter das letzte Element. Stand der Cursor vor dem ersten bzw. hinter dem letzten Element, so wird er durch *gotoNext* auf das erste Element gesetzt.

bool **gotoPrev**()

Setzt den Cursor auf das vorherige Element, das somit zum aktuellen wird. Stand der Cursor auf dem ersten Element, so gibt die Funktion den Wert false zurück und positioniert den Cursor vor das erste bzw. hinter das letzte Element. Stand der Cursor vor dem ersten bzw. hinter dem letzten Element, so wird er durch *gotoPrev* auf das letzte Element gesetzt.

bool **containsValue**(const value_type&) const

gibt *true* zurück, wenn ein Wert *s* im Container enthalten ist.

bool **findValue**(const value_type& xValue)

sucht nach dem ersten Wert xValue. Ist xValue als Wert vorhanden, wird das entsprechende Element zum aktuellen Element und die Funktion gibt *true* zurück. Andernfalls wird *false* zurückgegeben.

Beispiel:

```
vstr v;
vstr::cursor cr = v.getCursor();
if ( ! cr.findNext("tt") )
  out << "v is empty";
```

bool **findNextValue**(const value_type& xValue)

Durchsucht ausgehend von der aktuellen Position alle nachfolgenden Elemente nach einem Element, dessen Wert gleich xValue ist. Wird eine solches Element gefunden, wird dieses zum aktuellen Element und es wird *true* zurückgegeben, Falls kein solches Element gefunden wurde, bleibt die aktuelle Position unverändert und es wird *false* zurückgeliefert.

bool **findPrevValue**(const value_type& xValue)

Durchsucht ausgehend von der aktuellen Position die vorausgehenden Elemente nach einem Element, dessen Wert gleich xValue ist. Wird eine solches Element gefunden, wird dieses zum aktuellen Element und es wird *true* zurückgegeben, Falls kein solches Element gefunden wurde, bleibt die aktuelle Position unverändert und es wird *false* zurückgeliefert.

Für einen Map-Cursor sind einige zusätzliche Methoden definiert:

str **key**() const
bool **containsKey**(const str& xsKey) const
bool **findKey**(const str& xsKey)

10.3.3.4.4 Funktionstabelle

Eine Funktionstabelle ist im Prinzip eine Map, in der Funktionen nach String-Schlüsseln sortiert sind. Alle Funktionen einer Liste haben den gleichen Typ. Es gibt jedoch verschiedene Funktionstabellen-Typen, je nach dem Typ den die Funktionen zurückliefern. (Siehe obige Tabelle)

Syntax

mstrfun <bezeichner> ;
bool_mstrfun <bezeichner> ;
int_mstrfun <bezeichner> ;
uint_mstrfun <bezeichner> ;
dbl_mstrfun <bezeichner> ;
str_mstrfun <bezeichner> ;
node_mstrfun <bezeichner> ;
dnode_mstrfun <bezeichner> ;

Beschreibung

Diese sehr speziellen Container erleichtern die Verarbeitung von Parse-Bäumen, indem sie jedem **Label** eines Knotens eine **Funktion** zuordnen, die den Knoten auswerten kann.
Exemplarisch sei eine Funktionstabelle an Hand von *mstrfun* erläutert.So ist *mstrfun* einer map *mstrstr* ähnlich. *mstrfun* besitzt einen *str* als Schlüssel und einen weiteren *str* als Wert. Der Wert bezeichnet hier den Namen einer Klassenmethode.
Klassenmethoden, die über ihre Namen in eine Funktionstabelle eingefügt sind, müssen alle den **gleichen Typ** haben. Der Typ einer Funktion ergibt sich aus ihrem Rückgabetyp und ihren Parametern. So müssen alle in der Tabelle enthaltenen Funktionen den gleichen Rückgabetyp und die gleiche Anzahl und Art von Parametern haben. Eine zusätzliche Bedingung ist, dass der **erste Parameter** vom Typ: ***const node&*** ist. Mit diesem Parameter wird der Knoten übergeben, der ausgewertet werden soll.
Zur Definition einer mstrfun-Variable auf der Element-Seite des TextTransformers erscheint daher nach Eingabe des mstrfun-Typs das **Parameterfeld**, das sonst nur für die Definition von Funktionen erforderlich ist. Hier müssen nun die Parameter auf gleiche Weise spezifiziert werden, wie bei den einzufügenden Klassenmethoden.
Der **Rückgabetyp** der in der Tabelle enthaltenen Funktionen wird durch den speziellen Typ der Funktionstabelle bestimmt:

Typ der Funktionstabelle	**Rückgabetyp**
mstrfun	void
bool_mstrfun	bool
int_mstrfun	int
uint_mstrfun	unsigned int
dbl_mstrfun	double
str_mstrfun	str
node_mstrfun	node

bool **add**(LABEL, FUNCTION);

Im Textfeld, das für Variablen die **Initialisierungs-Routine** enthält, werden dann mittels des ***add* -Befehls** die einzelnen Funktionen eingefügt. Sämtliche Funktionen, die mstrfun enthalten soll, müssen bereits hier eingefügt werden. Es ist nicht möglich, in einer laufenden Transformation Funktionstabellen dynamisch zu erzeugen.
LABEL und FUNCTION können entweder als Strings - z.B. "number" - oder als Bezeichner - *number* - übergeben werden. Die Parameter als Bezeichner zu übergeben hat den Vorteil, dass das Syntax-Highlighting dann aktiv ist, und durch Anklicken mit der Maus zur entsprechenden Funktion gewechselt werden kann.
Bei der Erzeugung von exportierten C++-Code werden die Bezeichner automatisch in Strings bzw.

Funktionsadressen umgewandelt.
Es ist nicht möglich allgemeine Ausdrücke für diese Parameter zu verwenden.

Beispiel:

m_Eval sei eine mstrfun von void-Funktionen mit nur einen *node*-Parameter, wie

```
void VisitVariable(const node& xNode)
```

Der ersten beiden Aufrufe sind korrekt, der dritte aber nicht:

```
m_Eval.add( "Variable", "VisitVariable");
m_Eval.add( Variable, VisitVariable);

str sLabel =  "Variable"; m_Eval.add( sLabel, "VisitVariable"); // Fehler
```

Default-Funktion

Jede Funktionstabelle muss eine Default-Funktion enthalten, die diejenigen Knoten-Werte behandelt, deren Label mit keinem der Schlüssel der Tabelle übereinstimmt. Die Default-Funktion wird mit dem Leerstring "" als Schlüssel eingefügt:

Fortsetzung des obigen Beispiels:

```
void Default(const node& xNode)
m_Eval.add( "", "Default");
```

Die Erweiterung von Funktions-Tabellen durch neue Funktionen wird erleichtert durch den Funktions-Tabellen-Assistenten

. **visit**(const node& xNode ...)

Der Aufruf einer Funktion der Tabelle erfolgt durch einen Aufruf der *visit*-Methode der Tabelle selbst. Je nach dem Wert, den das Label des übergebenen Knotens hat, wird der *visit*-Aufruf automatisch in einen Aufruf der Methode umgeleitet, die in der Tabelle dem Wert zugeordnet ist.

Fortsetzung des obigen Beispiels:

```
node n( "Variable", ...);
m_Eval.visit( n );
```

ist äquivalent zu

```
if ( n.label() == "Variable")
    VisitVariable( n );
 else
   Default( n );
```

Im exportierten C++Code entspricht mstrfun die Klasse CTT_Mstrfun.

10.3.3.5 node / dnode

Syntax

node <bezeichner> ;
dnode <bezeichner> ;

Beschreibung

Die Typen *node* und *dnode* bezeichnen die Datenstruktur eines Baumknotens.Ein solcher Knoten enthält einen String - das Label - zur spezielleren Kennzeichnung des Knotentyps und einen weiteren String als eigentlicher Wert des Knotens. Schließlich kann dem Knoten noch ein Wert als unsigned int zugewiesen werden. Eine besondere Eigenschaft der Knoten besteht darin, dass sie miteinander verknüpfbar sind und so zusammen einen Baum bilden können.

node und *dnode* verfügen nahezu über die gleichen Schnittstellen und sind daher analog zu verwenden. Der genaue Unterschied ist weiter unten erläutert.

Entsprechend ihren Eigenschaften können *nodes* im Texttransformer in zweifacher Weise verwendet werden:

- als Container zum speichern von einzelnen Daten
- als Parse-Trees zur Repräsentation der grammatischen Struktur des gesmten Eingabetextes

Die Bezeichnungen und Relationen der Knoten sind im Glossar erklärt.

Node-Instanzen haben besondere Eigenschaften: sie sind referenz-gezählte Zeiger. Wird ein Knoten einem anderen zugewiesen, so wird bei Änderung des Werts (oder Labels) des einen, der Wert (bzw. das Label) des anderen zugleich mit verändert.

Beispiel.:

```
node n1("label1", "value1");
node n2("label2", "value2");
n1 = n2;
n1.setLabel("label3");
// jetzt haben sowohl n1 als auch n2 das Label "label3" und den Wert "value2"
```

Ein Baum existiert so lange, wie mindestens eine Referenz auf einen seiner Knoten existiert.

Die *node*-Funktionen sind in folgenden Kapiteln aufgelistet:

Suche
dnode Besonderheiten

10.3.3.5.1 node: Konstruktion

Funktionen zur Konstruktion von node-Objekten

Einzelne Knoten können mit oder ohne Initialisierung-Parameter erzeugt werden:

node()
node(str& xsLabel)
node(str& xsLabel, str& xsValue)

Eine neues node-Objekt wird erzeugt und je nach Parametern wird das Label mit *xsLabel* und er Wert mit *xsValue* initialisiert.

node(**const node**& xOther)

Eine **Referenz** auf das node-Objekt xOther wird erzeugt. D.h. der neue Knoten hat gleiches Label, gleichen Wert und dieselben Kinder wie das bisherige. Wird einer der beiden Knoten verändert, so ändert sich der zweite mit.

Die (vom Baum isolierte) Kopie eines Knotens erhält man durch:

node clone()

Beispiel:

```
node  nCopy  =  root_node.bottomLastChild().clone();
```

Um einen Knoten einer Baumstruktur hinzuzufügen, wird eine der folgenden Funktionen aufgerufen:

bool addChildFirst(**const node**& xNewChild)
bool addChildLast(**const node**& xNewChild)
node add(**const str**& xsLabel, **const str**& xsValue)

Durch die erste Funktion wird der neue Knoten zum ersten Kind-Knoten desjenigen Knotens, für den die Funktion *addChildFirst* aufgerufen wurde. Durch den Aufruf von *addChildLast* wird der übergebene Knoten zum letzten Kind-Knoten. Es wird jeweils *true* zurückgegeben, wenn das Einfügen des Knotens erfolgreich war.
Die dritte Funktion *add* kann als Akürzung gelesen werden für:

```
addChildLast( node( xsLabel, xsValue )
```

d.h. ein neuer Knoten mit dem Label xsLabel und dem Wert xsValue wird als letzter Kind-Knoten eingefügt. Zurückgegeben wird der neue Knoten oder node::npos, falls das Einfügen mißlang.

bool addChildBefore(**const node**& xnNewChild, **const node**& xnRefChild)

Mit dieser Funktion wird der neue Knoten *xnNewChild* vor dem vorhandenen Knoten *xnRefChild* eingefügt. Falls *xnRefChild* d/node::npos ist, wird der Knoten als letzter Kind-Knoten eingefügt. Wenn *xnNewChild* bereits in einem Baum enthalten war, wird er zunächst aus diesem entfernt. Wenn *xnNewChild* Kindknoten hat, werden sie mit transportiert.

node removeChild(**const node**& xnOldChild)

Ein vorhandener Kind-Knoten *xnOldChild* kann mit dieser Funktion aus dem Baum entfernt werden. Sie liefert den entfernten Knoten zurück. Wenn *xnOldChild* kein Kindknoten ist, wird *node::npos* zurückgeliefert.

node detach_node(**const node**& xn)

Diese globale Funktion vereinfacht den Gebrauch von *removeChild*, da sie nicht für den *Parent* -Knoten aufgerufen werden muss. Die Funktion ist definiert als:

```
if(xn.parent()  != node::npos)
    xn.parent().removeChild(xn);
return xn;
```

bool replaceChild(**node**& xNewChild, **node**& xOldChild)

Ersetzt den vorhandenen Kindknoten *xOldChild* durch den neuen *xNewChild*;

node und dnode verhalten sich verschieden, wenn versucht wird einen Knoten, der bereits in einen Baum eingefügt wurde, nochmals an anderer Stelle einzufügen:

Bis zur Version 1.7.2 war dies für einen **node**-Knoten **verboten**. und wurde mit einer Fehlermeldung quittiert. An Version 1.7.3 ist das verhalten an das der dnode's angeglichen.

Für einen **dnode**-Knoten bedeutet diese Operation eine **Verschiebung**: der Knoten samt seinen Unterknoten wird an der gewünschten Stelle eingefügt, verschwindet aber an der alten Position. Dies kann durchaus nützlich sein.

Die Speicherverwaltung von *node's* und *dnode*'s ist unterschiedlich: *node's* werden intern referenz-gezählt und *dnodes* werden extern durch das XMLDocument verwaltet. Der Verzicht auf ein solche "factory" bedeutet mehr Aufwand bei der Entfernung eines *node* aus einem Baum.

10.3.3.5.2 node: Information

Informationen über die Daten des Knotens (node) selbst oder über seine Stellung innerhalb eines Baums regeben die folgenden Funktionen.

str label() const

```
void          setLabel(const str& xsLabel)
str           value() const
void          setValue(const str& xsValue)
unsigned int  id() const
void          setId(unsigned int xuiId)

bool          hasChildren() const
bool          isDescendant( const node& xNode ) const
bool          isAncestor( const node& xNode ) const
bool          isSibling( const node& xNode ) const
unsigned int  level() const
unsigned int  descendentsCount() const
unsigned int  childCount() const

void          setAttrib(const str& xsLabel, const str& xsValue)
str           attrib(const str& xsLabel)
bool          hassAttrib() const
```

str **label**() const

Gibt den str-Wert des Labels zurück.

void **setLabel**(const str& xsLabel)

Setzt den String xsLabel als Label-Wert des Knotens.

str **value**() const

Gibt den str-Wert des Knotens zurück.

void **setValue**(const str& xsValue)

Setzt den String xsValue als Wert des Knotens.

unsigned int **id**() const
void setId(unsigned int xuiId)

Diese beiden Funktionen sind für die Weiter-Enwicklung des TextTransformers reserviert. Eventuell wird der Id-Typ noch geändert.

bool **hasChildren**() const

Die Funktion gibt *true* zurück, wenn der Knoten Kind-Knoten besitzt, andenfalls wird false zurückgeliefert.

bool **isDescendant**(const node& xNode) const

Mit *isDescendant* kann festgestellt werden, ob der übergebene Knoten *xNode* zum Zweig gehört, der in dem aktuellen Knoten seinen Ursprung hat.

bool **isAncestor**(const node& xNode) const

Mit *isAncestor* kann festgestellt werden, ob der aktuelle Knoten zum Zweig gehört, der in dem übergebenen Knoten *xNode* seinen Ursprung hat.

bool **isSibling**(const node& xNode) const

Mit *isSibling* kann festgestellt werden, ob der übergebene Knoten *xNode* in der Reihe der Geschwisterknoten auf den aktuellen Knoten folgt.

unsigned int **level**() const

Diese Funktion gibt für Knoten der obersten Ebene den Wert 0 an; die unmittelbar untergeordneten Objekte haben den Wert 1 usw.

unsigned int **descendentsCount**() const

Diese Funktion gibt die Anzahl der Knoten zurück, die sich in dem Zweig befinden, der in dem aktuellen Knoten seinen Ursprung hat.

unsigned int **childCount**() const

Diese Funktion gibt die Anzahl der Knoten zurück, die dem aktuellen Knoten unmittelbar untergeordnet sind.

Attribute

Attribute bestehen aus einem Schlüssel und einem zugehörigen Wert. Attribute sind z.B. nützlich zur Speicherung von Ini-Einträgen: während die Abschnitte der Ini-Datei durch Knoten repräsentiert werden, können die Werte des Abschnitts als Attribute gesetzt werden.

void **setAttrib**(const str& xsLabel, const str& xsValue)

Für jeden Knoten können mit *setAttrib* beliebig viele Attribute gesetzt werden. Ein Attribut hat einen Namen *xsLabel* und einen dazugehörigen Wert *xsValue*. Ist der Name noch nicht in der Liste der Attribute vorhanden, wird das Wertepaar neu eingefügt. Ist den Name bereits in der Liste vorhanden, so wird der zugehörige Wert mit dem neuen Wert überschrieben.

str **attrib**(const str& xsLabel)

Mit *attrib* kann der Wert des Attributs mit dem Namen *xsLabel* gelesen werden.

bool **hasAttrib**() const

Gibt *true* zurück, wenn der Knoten mindestens ein Attribut hat. Andernfallls wird *false* zurückgegeben.

Beispiel:

a) Ini parsen:

```
{{node n = xIni.add("[FONT]", ""); }}

"NAME"  "=" Value    {{ n.setAttrib("NAME", State.lp_str()); }}
"COLOR" "=" Value    {{ n.setAttrib("COLOR", State.lp_str()); }}
```

b) Werte auslesen:

```
node n = Ini.findNextLabel("[FONT]")

SetValues(n.attrib("NAME"), n.attrib("COLOR"));
```

10.3.3.5.3 node::npos

Ein besonderer Knoten (node) ist: *node::npos.* Er ist vergleichbar mit str::npos oder einem NULL-Zeiger in C++. Von allen Funktionen, die ein *node*-Objekt zurückgeben, wird *node::npos* immer dann zurückgegeben, wenn es den gewünschten Knoten nicht gibt. Z.B. ein neu erzeugter Knoten hat weder Kind- noch Eltern-Knoten.

```
node nNew;
node nPos = nNew.firstChild();
// nun gilt: nPos == node::npos
```

Bevor mit einem Knoten etwas angefangen wird, der durch eine der als Resultat einer Nachbar- oder Such-Funktion erhalten wurde, sollte daher stets verglichen werden, ob er node::npos ist oder nicht:

```
if(nPos != node::npos)
{
  // tu etwas mit nPos
}
else
 // tu nichts mit nPos
```

Sämtliche Knoten.Funktionen, die einen weiteren Knoten zum Resultat haben, geben auf *node::npos* angewandt wieder *node::npos* zurück. *node::npos* lässt sich nicht in einen Baum einfügen.

10.3.3.5.4 node: Nachbarn

Ausgehend von einem Knoten (node) können andere Knoten ermittelt werden, die in bistimmter Stellung zu ihm stehen:

```
node root() const
node parent() const

node firstChild() const
node lastChild() const

node nextSibling() const
node prevSibling() const
node firstSibling() const
node lastSibling() const

node bottomFirstChild() const
node bottomLastChild() const

node next() const
node follow() const
node prev() const
node nextLeaf() const
node prevLeaf() const
```

Wenn die Bezeichnungen und Relationen der Knoten, die im Glossar erklärt sind, bekannt sind, sind die Namen der meisten der obigen Funktionen selbsterklärend. Jede Funktion liefert für einen Knoten denjenigen Nachbarknoten zurück, der dem Funktionsnamen entspricht.

Beispiel:

```
{{
  node root("label_00", "value_00");
  root.add("label_11", "value_11");
  root.add("label_12", "value_12");

  node pos = root.firstChild();
  while(pos != node::npos)
  {
    out << "label: " << pos.label() << ", "
        << "value: " << pos.value() << endl;
    pos = pos.nextSibling();
  }
}}

ergibt:

label: label_11, value: value_11
label: label_12, value: value_12
```

Nach Ausführung dieser Zuweisung ist nChild gleich dem ersten Kind-Knoten des Knotens nParent.

Falls nParent keinen Kind-Knoten besitzt, gilt

nChild == node::npos

Weiter erläuterungsbedürftig sind:

node bottomFirstChild() const

Erstes Kind des ersten Kindes ...

node bottomLastChild() const

Unterstes letztes Kind des letzten Kinds

node next() const
node prev() const

next liefert in absteigender Richtung den nächsten Knoten zurück. *prev* liefert in aufsteigender Richtung den nächsten Knoten zurück.

node follow() const

follow liefert in absteigender Richtung den nächsten Knoten zurück, der auf den letzten Kind-Knoten folgt. Das ist entweder *nextSibling* des aktuellen Knotens oder *nextSibling* des ersten *parent* -Knotens, zu dem es ein *nextSibling* gibt oder node::npos.

node nextLeaf() const
node prevLeaf() const

nextLeaf liefert in absteigender Richtung den nächsten Knoten zurück, der keine Kindknoten hat. *prevLeaf* liefert in aufsteigender Richtung den nächsten Knoten zurück, der keine Kindknoten hat.

10.3.3.5.5 node: Suche

Ausgehend von einem Knoten (node) können andere Knoten mit bestimmten Eigenschaften gesucht werden:

```
node  findNextLabel(const str& xsLabel) const
node  findNextLabel(const str& xsLabel, const node& xnLast) const
node  findNextValue(const str& xsValue) const
node  findNextValue(const str& xsValue, const node& xnLast) const
node  findNextId(unsigned int xuiId) const
node  findNextId(unsigned int xuiId, const node& xnLast) const

node  findPrevLabel(const str& xsLabel) const
node  findPrevLabel(const str& xsLabel, const node& xnLast) const
```

node findPrevValue(const str& xsValue) const
node findPrevValue(const str& xsValue, const node& xnLast) const
node findPrevId(unsigned int xuiId) const
node findPrevId(unsigned int xuiId, const node& xnLast) const

node findChildLabel(const str& xsLabel, bool xbRecursive = true)
node findChildValue(const str& xsValue, bool xbRecursive = true)
node findChildId(unsigned int xuiId, bool xbRecursive = true)

node findParentLabel(const str& xsLabel)
node findParentValue(const str& xsValue)
node findParentId(unsigned int xuiId)

node findNextLabel(const str& xsLabel) const
node findNextLabel(const str& xsLabel, const node& xnLast) const

Liefert in absteigender Richtung den nächsten Knoten zurück, dessen Label gleich dem übergebenen String xsLabel ist. Existiert kein solcher Knoten ergibt die Funktion node::npos.
Mit dem zweiten optionalen Parameter *xnLast* kann ein Baumknoten bestimmt werden, an dem die Suche beendet wird.

Beispiel:

```
node pos = xn.findNextLabel( "parameter" );

while( pos != node::npos )
{
  out << "parameter: "   << pos.value() << endl;
   pos = pos.findNextLabel( "parameter" );
}
```

Diese Funktion gibt sämtliche Parameter aller möglichen Parameterlisten aus, die auf xn folgen. Sollen hingegen nur die Parameter einer aktuellen Liste ausgegeben werden, wäre zu schreiben:

```
node last = xn.bottomLastChild();
node pos = xn.findNextLabel( "parameter" , last);

while( pos != node::npos )
{
  out << "parameter: "   << pos.value() << endl;
   pos = pos.findNextLabel( "parameter", last );
}
```

node findNextValue(const str& xsValue) const
node findNextValue(const str& xsValue, const node& xnLast) const

Liefert in absteigender Richtung den nächsten Knoten zurück, dessen Wert gleich dem übergebenen String xsValue ist. Existiert kein solcher Knoten ergibt die Funktion node::npos.
Mit dem zweiten optionalen Parameter *xnLast* kann ein Baumknoten bestimmt werden, an dem die Suche beendet wird.

node findPrevLabel(const str& xsLabel) const
node findPrevLabel(const str& xsLabel, const node& xnLast) const

Liefert in aufsteigender Richtung den nächsten Knoten zurück, dessen Label gleich dem übergebenen String xsLabel ist. Existiert kein solcher Knoten ergibt die Funktion node::npos.
Mit dem zweiten optionalen Parameter *xnLast* kann ein Baumknoten bestimmt werden, an dem die Suche beendet wird.

node findPrevValue(const str& xsValue) const
node findPrevValue(const str& xsValue, const node& xnLast) const

Liefert in aufsteigender Richtung den nächsten Knoten zurück, dessen Wert gleich dem übergebenen String xsValue ist. Existiert kein solcher Knoten ergibt die Funktion node::npos.
Mit dem zweiten optionalen Parameter *xnLast* kann ein Baumknoten bestimmt werden, an dem die Suche beendet wird.

node findChildLabel(const str& xsLabel, bool xbRecursive = true)

Durchsucht die Menge aller Kind-Knoten nach dem nächsten Knoten, dessen Label gleich *xsLabel* ist. Falls ein solcher Knoten gefunden wird, wird er zurückgegeben, andernfalls wird node::npos zurückgeliefert.
Wenn *xbRecursive true* ist werden auch alle Kindeskind-Knoten durchsucht.

node findChildValue(const str& xsValue, bool xbRecursive = true)

Durchsucht die Menge aller Kinder- und Kindeskind-Knoten nach dem nächsten Knoten, dessen Wert gleich *xsValue* ist. Falls ein solcher Knoten gefunden wird, wird er zurückgegeben, andernfalls wird node::npos zurückgeliefert.
Wenn *xbRecursive true* ist werden auch alle Kindeskind-Knoten durchsucht.

node findParentLabel(const str& xsLabel)

Durchsucht die Menge aller Elternknoten nach dem nächsten Knoten, dessen Label gleich xsLabel ist. Falls ein solcher Knoten gefunden wird, wird er zurückgegeben, andernfalls wird node::npos zurückgeliefert.

node findParentValue(const str& xsValue)

Durchsucht die Menge aller Elternknoten nach dem nächsten Knoten, dessen Wert gleich xsValue ist. Falls ein solcher Knoten gefunden wird, wird er zurückgegeben, andernfalls wird node::npos zurückgeliefert.

node findNextId(unsigned int xuiId) const
node findNextId(unsigned int xuiId, const node& xnLast) const
node findPrevId(unsigned int xuiId) const
node findPrevId(unsigned int xuiId, const node& xnLast) const
node findChildId(unsigned int xuiId, bool xbRecursive = true)
node findParentId(unsigned int xuiId)

Analog zu den vorgenannten Funktionen kann mit diesen Funktionen nach einem bestimmten ID-Wert gesucht werden.

10.3.3.5.6 node: Sortierung

```
void    sortCildrentA()
void    sortCildrenD()
```

Durch Aufruf dieser Methoden werden die unmittelbaren Kindknoten sortiert. *sortCildrentA* sortiert die Knoten nach ihren Labels in alphabetisch aufsteigender (**a**scending) Ordnung und *sortCildrenD* sortiert in alphabetisch absteigender (**d**escending) Ordnung.

10.3.3.5.7 dnode Besonderheiten

node und *dnode* verfügen nahezu über die gleichen Schnittstellen und sind daher analog zu verwenden. Im Unterschied zu *node* gehört *dnode* zu einem xerces Dokument. Solche Dokumente eröffnen einen großen Spielraum zur Manipulation im erzeugten C++-Code und sie können leicht als XML-Dokumente geschrieben werden. Einige dieser Erweiterungen sind auch im Interpreter verfügbar. Bäume aus *dnode*-Knoten werden im Unterschied zu solchen aus *nodes* im Variablen-Inspektor auch noch angezeigt, nachdem eine Transformation beendet ist. Das liegt daran, dass das Dokument für die *dnodes* zum Plugin gehört, das auch außerhalb des Parsers existieren kann, während *nodes* nur lokal innerhalb des Parsers existieren. Dies ist auch der Grund, warum *dnode* Klassenelemente nicht vor Beginn des Parsens initialisiert werden können. Erst dann wird das Plugin in den Parser hinein gereicht, so dass die *dnode* Knoten vom DOMDocument des Plugins erzeugt werden können. Bei der Erzeugung von C++ Code, der *dnodes* verwendet, muss die Xerces-Bibliothek mit gelinkt werden. In den Projekt-Optionen für die Code-Erzeugung ist dann CTT_ParseStateDomPlugin als Plugin-Typ zu setzen. Die Auswertung mittels Funktionstabellen im exportierten Code ist noch nicht implementiert

Der Wurzelknoten eines *dnode*-Baums muss mit der Funktion *GetDocumentElement* erzeugt werden. Geschrieben werden kann das Dokument schließlich mit der Funktion *WriteDocument*

Die Art und Weise, wie das Dokument geschrieben wird, ist in den Projektoptionen festgelegt.

Während Label und Wert einer *node* aus beliebigen Zeichen bestehen können,.ist die **Menge der Zeichen für die Label einer *dnode*** auf Buchstaben, Ziffern und den Unterstrich eingeschränkt, wobei das Label nicht mit einer Ziffer oder dem Unterstrich beginnen darf. Sonderzeichen und Umlaute sind hier nicht erlaubt. Erlaubt sind genau die Zeichen, die auch für die Definition eines XML-tags erlaubt sind.

Als weiterer Unterschied von *node* und *dnode* sei nochmals darauf hingewiesen, dass *dnode*-Zweige im Unterschied zu Zweigen aus *node's* im Baum verschoben werden können.

10.3.3.6 const

Syntax

const <variablenname> [= <wert>] ;
<funktionsname> (const <typ>*<variablenname> ;)
<funktionsname> const;

Beschreibung

Der Modifizierer const wird verwendet, um Änderungen des Werts einer Variable zu verhindern.

Verwenden Sie den Modifizierer const, um einer Variablen einen vom Programm nicht mehr veränderbaren Anfangswert zu geben. Jede spätere Wertzuweisung für eine const-Variable führt zu einer Fehlermeldung des Compilers.

10.3.3.7 Operatoren

Auf die elementaren Datentypen und Strings und zum Teil auch auf andere Typen können Operatoren angewendet werden:

Arithmetische Operatoren
Zuweisungsoperatoren
Relationale Operatoren
Gleichheitsoperatoren
Logische Operatoren

10.3.3.7.1 Arithmetische Operatoren

Es gibt folgende arithmetische Ooperatoren:
Binäre Operatoren:

Op1 * Op2
Op1 / Op2
Op1 % Op2 (Modulus, Rest)
Op1 + Op2
Op1 - Op2

Unäre Operatoren:
Op++
Op1--

Bei Typen, die Zahlen repräsentieren bewirken die Operatoren + (Addition), - (Subtraktion), * (Multiplikation) und / (Division) die Ausführung der jeweiligen Grundrechenart.
(op1 % op2) ergibt den Rest bei der Division (op1 dividiert durch op2)

Für die Operatoren / und % muß op2 ungleich Null sein, op2 = 0 ergibt einen Fehler (man kann nicht durch Null teilen).
% kann nicht für den double-Typ verwendet werden.

Der Operator ++ (Inkrement) addiert die Zahl 1 zum Wert des Ausdrucks.
Der Operator -- (Dekrement) subtrahiert die Zahl 1 vom Wert des Ausdrucks.

Bei Strings bewirkt die Addition eine Verkettung der Zeichen.

10.3.3.7.2 Zuweisungsoperatoren

Es gibt folgende Zuweisungsoperatoren:

= *= /= += -= %= ^= |= &= <<= >>=

Der Wert des Operanden Op1 nach Ausführung der Zuweisung

Op1 = Op2;

ist gleich dem Wert von Op2.

Der Ausdruck

Op1 op= Op2;

hat den gleichen Effekt wie

Op1 = Op1 op Op2;

Beispiel: Op1 += Op2; ist gleichwertig mit Op1 = Op1 + Op2;.

Die Operanden Op1 und Op2 müssen entweder vom gleichen Typ sein oder zueinander kompatibel sein.

Für Op2 kann der Aufruf einer Produktion oder eines Tokens stehen, wenn diese einen Wert zurückliefern und die schließende Klammer der semantischen Aktion unmittelbar auf die Operation folgt. Beispiele:

```
{{e = }} Term
{{e += }} Term
{{e -= }} Term
```

10.3.3.7.3 Relationale Operatoren

Syntax

relationaler-ausdruck < schiebe-ausdruck
relationaler-ausdruck > schiebe-ausdruck
relationaler-ausdruck <= schiebe-ausdruck
relationaler-ausdruck >= schiebe-ausdruck

Bemerkungen:

Die relationalen Operatoren werden für die Prüfung von Ausdrücken auf Gleichheit oder Ungleichheit benutzt. Wenn die Aussage wahr ist, ergibt der relationale Ausdruck den Wert Wahr (1), andernfalls Falsch (0).

> größer als
< kleiner als

>= größer als oder gleich
<= kleiner als oder gleich

In dem Ausdruck

E1 <operator> E2

müssen die Operanden E1 und E2 arithmetische Typen sein.

10.3.3.7.4 Gleichheitsoperatoren

Es gibt zwei Gleichheitsoperatoren: **==** und **!=**. Sie überprüfen die Gleichheit bzw. die Ungleichheit von arithmetischen Werten oder strings. Die Anwendungsregeln sind denen für die relationalen Operatoren sehr ähnlich.

Hinweis:
Beachten Sie, daß == und != eine niedrigere Abarbeitungspriorität haben als die relationalen Operatoren <, >, <=, und >=. Mit == und != können Sie auch strings auf Gleichheit und Ungleichheit miteinander vergleichen, bei denen die relationalen Operatoren nicht erlaubt sind.

Die **Syntax** lautet:

Gleichheitsausdruck

Relationaler-Ausdruck
Gleichheitsausdruck == Relationaler-Ausdruck

Gleichheitsausdruck != Relationaler-Ausdruck

10.3.3.7.5 Logische Operatoren

Syntax

logischer-UND-ausdruck && inklusiv-ODER-ausdruck
logischer-ODER-ausdruck || logischer-UND-ausdruck
! cast-ausdruck

Bemerkungen:

Operanden in logischen Ausdrücken müssen von einem skalaren Typ sein.

- **&&** Logisches UND; liefert das Ergebnis Wahr (1), wenn beide Ausdrücke bei der Auswertung einen Wert ungleich Null ergeben, andernfalls ist das Ergebnis Falsch (0). Wenn der erste Ausdruck Falsch ergibt, wird der zweite nicht ausgewertet.
- **||** Logisches ODER; iefert das Ergebnis Wahr (1), wenn mindestens einer der beiden Ausdrücke bei der Auswertung einen Wert ungleich Null ergibt, andernfalls ist das Ergebnis Falsch (0). Wenn der erste Ausdruck Wahr ergibt, wird der zweite nicht ausgewertet.

- **!** Logische Negation; liefert das Ergebnis Wahr (1), wenn der gesamte Ausdruck einen Wert von Null hat, andernfalls ist das Ergebnis Falsch (0). Der Ausdruck !E ist gleichwertig mit (0 ==

E).

10.3.3.7.6 Bitweise Operatoren

Bemerkung:

Die bitweisen Operatoren dienen zur Änderung einzelner Bits anstelle einer kompletten Zahl.

Operator	Beschreibung
&	Bitweises UND; vergleicht paarweise je zwei korrespondierende Bits und setzt das entsprechende Bit im Ergebnis auf 1, wenn beide Bits 1 sind, andernfalls auf 0.
|	Bitweises inklusives ODER; vergleicht paarweise je zwei korrespondierende Bits und setzt das entsprechende Bit im Ergebnis auf 1, wenn eins oder beide Bits 1 sind, andernfalls auf 0.
^	Bitweises exklusives ODER; vergleicht paarweise je zwei korrespondierende Bits und setzt das entsprechende Bit im Ergebnis auf 1, wenn beide Bits unterschiedlich sind, andernfalls auf 0.
~	Bitweise Negation; invertiert jedes Bit. Dieser Operator wird auch zum Erzeugen von Destruktoren benutzt.
>>	Bitweises Schieben nach rechts; verschiebt alle Bits nach rechts, wobei das jeweils ganz rechte Bit verworfen und wenn kein Vorzeichen vorhanden ist, das ganz linke auf 0 gesetzt wird, ansonsten wird das Vorzeichen übernommen.
<<	Bitweises Schieben nach links; verschiebt alle Bits nach links, wobei das jeweils ganz linke Bit verworfen und das ganz rechte auf 0 gesetzt wird.

Beide Operanden eines bitweisen Operators müssen einen Ganzzahltyp haben.

Bitwert E1	Bitwert E2	Ergebnis E1 & E2	Ergebnis E1 ^ E2	Ergebnis E1 | E2
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

Hinweis: Die Operatoren &, >> und << sind kontextabhängig.
& kann auch der Referenz-Operator sein.
>> kann auch der Eingabe-Operator in einem Ein-/Ausgabeausdruck sein.
<< kann auch der Ausgabe-Operator in einem Ein-/Ausgabeausdruck sein.

10.3.3.7.7 Bedingungsoperator

Syntax

logischer-OR-ausdruck ? ausdruck : conditional-ausdruck

Bemerkungen:

Der Bedingungsoperator ?: ist ein ternärer Operator.

Im Ausdruck E1 ? E2 : E3 wird E1 zuerst ausgewertet. Wenn sein Wert ungleich Null (Wahr) ist, so wird danach E2 ausgewertet und E3 ignoriert. Wenn E1 Null (Falsch) ergibt, so wird E3 ausgewertet und E2 ignoriert.

Das Ergebnis von E1 ? E2 : E3 ist entweder der Wert von E2 oder von E3, abhängig davon, welcher dieser Werte ausgewertet wurde.

Der Bedingungsoperator ist im TETRA-Interpreter nur von sehr eingeschränktem Nutzen, da für die E's z.Z. noch nicht das volle Spektrum von C++-Expressions zugelassen ist. Genaueres hierzu kann man dem Parser für den TETRA C++-Interpreter entnehmen.

10.3.3.8 Kontrollstrukturen

Der TETRA-Interpreter beherrscht die C/C++ Kontrollstrukturen:

if, else
for
while
do
switch

10.3.3.8.1 if, else

Syntax

if (<bedingung>) <anweisung1>;

if (<bedingung>) <anweisung1>;
else <anweisung2>;

Beschreibung

Die Anweisung **if** dient zum Programmieren einer Bedingungsabfrage.

Der Ausdruck <bedingung> muss sich zu einem Wert vom Typ bool auswerten lassen, andernfalls ist die Bedingung ungültig.

Wenn die Auswertung von <bedingung> den Wert true ergibt, so wird <anweisung1> ausgeführt.

Wenn <bedingung> false ergibt, so wird <anweisung2> ausgeführt.

Die Klausel else ist optional, es dürfen aber zwischen einer **if**-Anweisung und **else** keine weiteren

Anweisungen stehen.

Im Unterschied zum Standard C/C++ können innerhalb des bedingten Ausdrucks keine Variablen deklariert werden und keine Zuweisungen erfolgen. Folgendes ist daher nicht möglich:

```
if (int val = stoi("1")) oder
int val; if (val = stoi("1"))
```

Möglich ist hingegen:

```
if (stoi("1"))
```

10.3.3.8.2 for

Syntax

for ([<initialisierung>] ; [<bedingung>] ; [<inkrement>]) <anweisung>

Beschreibung

Die Anweisung **for** dient zum Programmieren einer Schleife.

<condition> wird vor dem ersten Einstieg in den Block geprüft.

<anweisung> wird wiederholt so oft ausgeführt, bis der Wert von <bedingung> false ist.

Vor dem ersten Durchlauf der Schleife werden die Variablen für die Schleife durch <initialisierung> initialisiert.
Nach jedem Durchlauf der Schleife wird die Laufvariable durch <inkrement> erhöht. (Deshalb ist ++j in diesem Fall funktionell gleichwertig mit j++.)

In C++ kann <initialisierung> ein Ausdruck oder eine Deklaration sein.

Der Gültigkeitsbereich der dort deklarierten Bezeichner erstreckt sich nur bis zum Ende der **for** -Anweisung.

Alle drei Ausdrücke sind optional. Wenn <bedingung> weggelassen wird, so wird dafür der Wert true angenommen.

10.3.3.8.3 while

Syntax

while (<bedingung>) <anweisung>

Beschreibung

Das Schlüsselwort **while** dient zur bedingten wiederholten Ausführung einer Anweisung.

Die <anweisung> wird so lange wiederholt ausgeführt, bis der Wert von <bedingung> false ist.

Die Prüfung findet statt, bevor <anweisung> ausgeführt wird. Deshalb wird die Schleife keinmal durchlaufen, falls <bedingung> zu Anfang des ersten Durchlaufs den Wert false ergibt.

10.3.3.8.4 do

Syntax

do <anweisung> **while** (<bedingung>);

Beschreibung

Die Anweisung **do** wird solange ausgeführt, bis die <bedingung> zu false ausgewertet wird.

<anweisung> wird solange wiederholt ausgeführt, wie der Wert von <bedingung> gleich true bleibt.

Da die Bedingung nach jeder Ausführung von <anweisung> geprüft wird, wird die Schleife mindestens einmal durchlaufen.

10.3.3.8.5 switch

Syntax

```
switch ( <switch_variable> ) {
  case <konstantenausdruck> : <anweisung>; [break;]
    .
    .
    .
  default : <anweisung>;
}
```

Beschreibung

Die Anweisung **switch** übergibt die Ablaufkontrolle an dasjenige **case**-Label, das dem Wert der <switch_variable> entspricht, und es werden dann die dem **case**-Label folgenden Anweisungen ausgeführt.

Wenn kein **case**-Label die Bedingung erfüllt, so geht die Ablaufkontrolle zum Label **default**, und die darauf folgenden Anweisungen werden ausgeführt.

Um zu vermeiden, dass anschließend noch die Anweisungen anderer **case**-Zweige ausgeführt werden und um die Ausführung der **switch**-Anweisung zu beenden, schließt man jeden **case**-Zweig mit der Anweisung **break**;ab.

10.3.3.9 Ausgabe

Das Ergebnis einer Transformation erscheint auf dem Bildschirm oder in einer Datei, wenn es dorthin geschrieben wird. Repräsentanten dieser Ausgabeziele sind

out	für die Resultate der Transformation
log	für Zusatzinformationen zu einer Transformation

Im Normalfall generiert der TextTransformer Textdateien. Für spezielle professionelle Anwendungen ist es auch möglich binäre Dateien zu erzeugen.

10.3.3.9.1 out

out steht für den Zieltext: innerhalb der IDE Arbeitsoberfläche erscheint dieser Text im Zielfenster, im Transformations-Managers, dem Kommandozeilen-Werkzeug und dem generierten C++-Parser erscheint der Zieltext in einer Datei. Um einen Text oder ein Variablen-Inhalt nach *out* zu schreiben, wird der "Shift"-Operator "<<" verwendet.

Beispiel:

Sei "sResult" eine str-Variable, die den Text "42" enthält, so bewirkt die Anweisung:

```
out << sResult;
```

daß der Text "42" im Ausgabefenster erscheint. Shift-Anweisungen lassen sich verketten:

```
out << "Das Ergebnis lautet: " << sResult << "\n";
```

Die letztere Anweisungskette läßt im Ausgabefenster den Text:

"Das Ergebnis lautet: 42"

erscheinen. (Das Zeichen "\n" steht für ein Zeilenende und bewirkt, das die darauffolgende Ausgabe in die nachfolgende Zeile geschrieben wird.)

Das Ergebnis des Aufrufs einer Produktion oder eines Tokens, die einen Wert zurückliefern kann direkt in die Ausgabe geschrieben werden. Hierzu läßt man die schließende Klammer der semantischen Aktion unmittelbar auf den Shift-Operator folgen, weiter unmittelbar gefolgt vom Aufruf der Regel. Beispiel:

```
{{ out << }} Rule
```

Anmerkung: Die *out*-Ausgabe kann mit RedirectOutput in eine andere Datei umgelenkt werden.

Anmerkung: Bei der Erzeugung von C++-Code wird *out* durch den Ausdruck: *xState.out()*, ersetzt. *xState.out()* liefert ein ostream-Objekt des Plugins.

endl

```
out << endl
```

ist eine andere Schreibweise für

```
out << '\n'
```

In C++ wird durch diese Anweisung zusätzlich der Ausgabepuffer geleert.

Anmerkung: Bis zur Version 0.9.8.6 wurde **cout** geschrieben statt *out*. *cout* steht in C++ für den Standard-Ausgabekanal, üblicherweise die Konsole.

10.3.3.9.2 log

Ähnlich wie nach out können Ausgaben auch nach **log** geschrieben werden. Im Unterschied zu out, erscheint die Ausgabe aber nicht im Zieltext sondern im Log-Fenster. Diese Ausgaben sind als Meta-Informationen zum Ablauf eines Programms gedacht und können bei der Suche nach eventuellen Fehlern helfen.

Beispiel:

```
log << "Wert: " << xs;
```

Wenn der Parser als C++-Code exportiert wird, werden die log-Anweisungen nicht ausgegeben, wenn nicht die Option aktiviert ist, den Code nur zu kopieren. Um auch bei dieser Option ausführbaren Programm-Code erzeugen zu können, ist es möglich statt "log" auch "clog" zu schreiben.

10.3.3.9.3 Binäre Ausgaben

Zum Schreiben von Variablen in binäre Dateien gibt es im TextTransformer eine einfache Schreibweise:

Beispiel:

```
out << int_bin( 42 ) << double_bin( 123,456 );
```

oder noch einfacher:

```
out << bin( 42 ) << bin( 123.456 );
```

Im exportierten C++-Code wird die zweite Schreibweise automatisch in die erste übersetzt. In diesen Ausdrücken wird jeweils ein überschriebener Ausgabeoperator für temporäre Objekte aufgerufen, die für das Schreiben der Binärformen der jeweiligen Variablentypen sorgen.

Die Variablentypen *bool*, *char*, *int*, *unsigned int*, *float* und *double* können auf diese Weise binär geschrieben werden. *string_bin* schreibt string::c_str() in die Ausgabe.

```
bool_bin( bool b )
```

```
int_bin( int i )
uint_bin( unsigned int ui )
float_bin( float f )
double_bin( double d )
char_bin( char c )
string_bin( string s )
```

Ein Spezialfall ist das Nullzeichen '\0'. Um dieses binär zu schreiben dieses gibt es neben *char_bin*('\0') den gesonderten Manipulator *ends*.

```
out << ends; // schreibt 00
```

Um zu verhindern, dass '\n' in "\r\n' umgewandelt wird, sollte die Datei im Binärmodus geöffnet werden.
Binäre Daten werden im Ausgabefenster des TextTransformers nur verstümmelt wiedergegeben.

10.3.3.10 return

Syntax

return <ausdruck> ;

Beschreibung

Die Anweisung return dient dazu, die aktuelle Funktion zu verlassen und zur aufrufenden Funktion zurückzukehren, wobei ein Wert zurückgeliefert wird.
Sowohl Produktionen, als auch Elementfunktionen als auch Tokenaktionen können werte zurückliefern.

Anmerkung: der Typ des zurückgelieferten Werts muss mit dem Rückgabetyp, der in dem entsprechenden Feld des Skripts deklariert ist, kompatibel sein.

10.3.3.11 break

Syntax

break;

Beschreibung

Die Anweisung break bewirkt innerhalb einer Schleife, daß die Ablaufkontrolle zur ersten Anweisung hinter der innersten Schleife, in der break steht, springt.

10.3.3.12 continue

Syntax

continue;

Beschreibung

Die Anweisung continue wird innerhalb von Schleifen benutzt, um die Steuerung an das Ende der innersten Schleife, die zu dem Schleifenkonstrukt (z.B. for oder while) gehört, springen zu lassen. An diesem Punkt wird dann die Fortsetzungsbedingung der Schleife erneut geprüft.

10.3.3.13 throw

Mit einer Anweisung der Art:

throw CTT_Error("Fehlermeldung");

kann ein TETRA-Progamm abgebrochen werden, wobei statt "Fehlermeldung" ein genauerer Text oder auch eine str-Variable übergeben werden. Im C++ Jargon sagt man für diese Anweisung, dass die Ausnahmeklasse oder Exception "CTT_Error" ausgeworfen wird. Innerhalb des TextTransformers wird der Text der Fehlermeldung in der Log-Fenster angezeigt.

Vom Benutzer programmierte Fehlerausgabe: Fehlermeldung.

Einen derartiger Programmabbruch erfolgt in der Term-Produktion des Rechner-Beispiels, zur Verhinderung einer Division durch Null.

Eine Alternative zur *throw*-Anweisung ist die Funktion *GenError*.
Es ist auch möglich einen Programmabbruch bei deaktiivertem Interpreter zu erzwingen. Zu diesem Zweck dient das Schlüsselwort EXIT.

10.3.4 String-Manipulation

Innerhalb des TETRA Interpreters sind einige Befehle zur String-Manipulation verfügbar, die nicht standardisiert sind.

stod	Konvertierung eines str nach double
stoi	Konvertierung eines str nach int
hstoi	Konvertierung eines hexadezimalen Strings nach int
stoc	Konvertierung eines Strings nach char
dtos	Konvertierung eines double-Werts in einen str
itos	Konvertierung eines int-Werts in einen str
itohs	Konvertierung eines int-Werts in einen str
ctohs	Konvertierung eines Zeichens in einen hexadezimalen str
ctos	Konvertierung eines Zeichens in einen str

Anmerkung: Eine Integer-Variable, die einen ANSI-Wert repräsentiert, kann direkt in das Zeichen konvertiert werden, indem sie einer char-Variablen zugewiesen wird.

```
char c = 65; //  c == 'A'
```

to_upper_copy	Gibt einen str in Großschreibung zurück
to_lower_copy	Gibt einen str in Kleinschreibung zurück

trim_left_copy	entfernt führende Leerzeichen
trim_right_copy	entfernt abschließende leerzeichen
trim_copy	entfernt Leerzeichen an den Enden eines str

In der Professional-Version wird eine Quelltextdatei dieser Befehle mitgeliefert: tt_lib.cpp

10.3.4.1 stod

Prototyp

```
double stod(const str& xs)
```

Beschreibung

Konvertiert einen *str* in eine Doublezahl.

Rückgabewert

stod liefert das Ergebnis der Konvertierung oder wirft eine Exception (boost::bad_lexical_cast), wenn der zu interpretierende String nicht zu einem Doublewert konvertierbar ist.

10.3.4.2 stoi

Prototyp

```
int stoi(const str& xs)
```

Beschreibung

Konvertiert einen *str* in eine Integerzahl.

Rückgabewert

stoi liefert das Ergebnis der Konvertierung oder wirft eine Exception (boost::bad_lexical_cast), wenn der zu interpretierende String nicht zu einem Integerwert konvertierbar ist.

10.3.4.3 hstoi

Prototyp

```
int hstoi(const str& xs)
```

Beschreibung

Konvertiert einen *str* aus Hexadizimal-Zeichen in eine Integerzahl.

Rückgabewert

hstoi liefert das Ergebnis der Konvertierung oder 0, wenn der zu interpretierende String nicht zu einem Integerwert konvertierbar ist.

Beispiel:

hstoi("FF") ergibt die Zahl 255.

10.3.4.4 stoc

Prototyp

char stoc(const str& xs)

Beschreibung

Konvertiert einen *str* in ein Zeichen.

Rückgabewert

Gibt das erste Zeichen des Strings *xs* zurück oder, falls der String leer ist, '\0'.

Beispiel:

```
char c = stoc("hello");
// c == 'h'
```

10.3.4.5 dtos

Prototyp

str dtos(double xd)

Beschreibung

Konvertiert eine Doublezahl in einen *str*.

Rückgabewert

dtos liefert das Ergebnis der Konvertierung oder wirft eine Exception (boost::bad_lexical_cast), wenn der Wert nicht zu einem str konvertierbar ist.

10.3.4.6 itos

Prototyp

str itos(int xi)

Beschreibung

Konvertiert eine Integerzahl in einen *str*.

Rückgabewert

itos liefert das Ergebnis der Konvertierung oder wirft eine Exception (boost::bad_lexical_cast), wenn der Wert nicht zu einem str konvertierbar ist.

10.3.4.7 itohs

Prototyp

str itohs(int xi)

Beschreibung

Konvertiert eine Integerzahl in einen *str* aus Hexadezimal-Zeichen.

Rückgabewert

itohs liefert einen String als Ergebnis der Konvertierung.

Beispiel:

itohs(1000) ergibt den String: "3e8".

10.3.4.8 ctohs

Prototyp

str ctohs(chat xc)

Beschreibung

Konvertiert ein Zeichen in einen *str* aus Hexadezimal-Zeichen.

Rückgabewert

ctohs liefert einen String als Ergebnis der Konvertierung.

Beispiel:

ctohs('A') ergibt den String: "41".

10.3.4.9 ctos

Prototyp

str ctos(char xc)

Beschreibung

Konvertiert ein Zeichen in einen *str*.

Rückgabewert

liefert einen String, der als einziges Zeichen *xc* enthält

Beispiel:

```
str s = "hello";
s = ctos( s[ 0 ] );
// s == "h";
```

10.3.4.10 to_upper_copy

Prototyp

str to_upper_copy(const str& xs)

Beschreibung

Gibt einen String zurück, der aus dem String xs erzeugt wird, indem sämtliche Kleinbuchstaben in Großbuchstaben umgewandelt werden . Zeichen, die bereits Großbuchstaben sind, werden dabei unverändert gelassen.

Beispiel:

```
str s = to_upper_copy("TextTransformer");
```

ergibt einen String s mit dem Inhalt: "TEXTTRANSFORMER".

10.3.4.11 to_lower_copy

Prototyp

str to_lower_copy(const str& xs)

Beschreibung

Gibt einen String zurück, der aus dem String xs erzeugt wird, indem sämtliche Großbuchstaben in Kleinbuchstaben umgewandelt werden. Zeichen, die bereits Kleinbuchstaben sind, werden dabei unverändert gelassen.

Beispiel:

```
str s = to_lower_copy("TextTransformer");
```

ergibt einen String s mit dem Inhalt: "texttransformer".

10.3.4.12 trim_left_copy

Prototyp

str trim_left_copy(const str& xs)

Beschreibung

Entfernt alle führenden Leerzeichen, Tabulatoren und Zeilenumbrüche aus xs.Das Ergebnis ist eine getrimmte Kopie von xs.

Beispiel:

```
str s = trim_left_copy("   TextTransformer   ");
```

hat zum Ergebnis: s == "TextTransformer   ".

10.3.4.13 trim_right_copy

Prototyp

str trim_right_copy(const str& xs)

Beschreibung

Entfernt alle Leerzeichen, Tabulatoren und Zeilenumbrüche am Ende von xs.Das Ergebnis ist eine getrimmte Kopie von xs.

Beispiel:

```
str s = trim_right_copy("   TextTransformer   ");
```

hat zum Ergebnis: s == "   TextTransformer".

Dieser Befehl eignet sich dazu, Text zu extrahieren, der von einem SKIP-Symbol abgedeckt ist. Hierbei werden zwar führende Leerzeichen ignoriert, nicht aber die Leerzeichen, die vor dem Token stehen, zu dem gesprungen wird. Beispielsweise :

SKIP {{**out** << **trim_right_copy**(**xState**.str()); }} "$"

ergibt für die Eingabe : " 77.74 $"
die Ausgabe : "77.74"

xState.str() == "77.74 " und trim_right_copy(xState.str()) == "77.74".

10.3.4.14 trim_copy

Prototyp

str trim_copy(const str& xs)

Beschreibung

Entfernt alle Leerzeichen, Tabulatoren und Zeilenumbrüche vom Anfang und am Ende von xs.Das Ergebnis ist eine getrimmte Kopie von xs.

Beispiel:

```
str s = trim_copy("   TextTransformer   ");
```

hat zum Ergebnis: s == "TextTransformer".

10.3.5 Dateibehandlung

Die folgenden Funktionen zur Pfad- und Dateibehandlung basieren auf der portablen boost filesystem Bibliothek.

basename	Liefert den Basisnamen eines Dateinamens
extension	Liefert die Erweiterung eines Dateinamens
change_extension	Ändert die Erweiterung eines Dateinamens
append_path	Setzt Pfade zusammen
current_path	Liefert den aktuellen Pfad
exists	Prüft die Existenz eines Pfades

is_directory	Prüft, ob der Pfad ein Verzeichnis bezeichnet
file_size	Liefert die Dateigröße
find_file	Sucht eine Datei in einem Verzeichnis
load_file	Lädt den Inhalt einer Datei
load_file_binary	Lädt den Inhalt einer Datei im Binärmodus
path_separator	String-Konstante für den Pfad-Trenner

siehe auch:Quelle und Ziel, Ausgabe-Umlenkung, Unit_dependence Beispiel

10.3.5.1 basename

Prototyp

str basename(**const** str& xsPath);

Beschreibung

Wenn im Pfad xsPath der Teil nach dem letzten Pfadtrenner einen Punkt ('.') enthält, wird der Teil bis zum Punkt (den Punkt nicht eingeschlossen) zurückgegeben. Sonst wird der gesamte Teil zurückgegeben.

Beispiel 1:

```
out << basename("D:\\TextTransformer\\Settings\\EditDefault.ds");
```

Ausgabe:

```
EditDefault
```

Beispiel 2:

```
out << basename("D:\\TextTransformer\\Settings");
```

Ausgabe:

```
Settings
```

10.3.5.2 extension

Prototyp

str extension(**const** str& xsPath);

Beschreibung

Wenn im Pfad xsPath der Teil nach dem letzten Pfadtrenner einen Punkt ('.') enthält, wird der Teil nach dem Punkt (den Punkt eingeschlossen) zurückgegeben. Sonst wird ein leerer String zurückgegeben.

Beispiel:

```
out << extension("D:\\TextTransformer\\Settings\\EditDefault.ds");
```

Ausgabe:

```
.ds
```

10.3.5.3 change_extension

Prototyp

str change_extension(**const** str& xsPath, **const** str& xsNewExtension);

Beschreibung

Diese Funktion gibt den Pfad zurück, der sich aus dem Austausch der Erweiterung im Pfad xsPath durch xsNewExtension ergibt. xsNewExtension sollte einen Punkt enthalten, um das vermutlich gewünschte Ergebnis zu liefern.

Beispiel 1:

```
out << change_extension("EditDefault.ds",  ".tb");
```

Ausgabe:

```
EditDefault.tb
```

Beispiel 2:

```
out << change_extension("EditDefault.ds",  "tb");
```

Ausgabe:

```
EditDefaulttb   // vermutlich nicht gewünscht
```

10.3.5.4 append_path

Prototyp

str append_path(**const** str& xsPath1, **const** str& xsPath2);

Beschreibung

Gibt einen aus den Teilen xsPath1 und xsPath2 zusammengesetzten Pfad zurück, in dem sie durch einen Pfad-Trenner verbunden sind. Ist xsPath1 oder xsPath2 leer, so wird der jeweils andere Teil zurückgegeben, ohne einen Pfad-Trenner anzufügen.

Beispiel:

```
str sLogPath = append_path ( current_path(),  "log.txt");
```

statt

```
str sLog = current_path() + path_separator + "log.txt";
```

Unter Windows gilt:

```
append_path("a", "b") == "a\\b"
append_path("a\\", "b") == "a\\b"
append_path("a", "\\b") == "a\\b"
append_path("a\\", "\\b") == "a\\b"
append_path("a", "") == "a"
append_path("", "b") == "b"
```

10.3.5.5 current_path

Prototyp

str current_path();

Beschreibung

Gibt den aktuellen Pfad zurück, der vom Betriebssystem verwaltet wird.

10.3.5.6 exists

Prototyp

bool exists(const str& xsPath);

Beschreibung

Gibt *true* zurück, wenn das Betriebssystem meldet dass der Pfad, der durch xsPath bezeichnet wird existiert. Andernfalls wird *false* zurückgegeben.

Beispiel:

```
if(exists(TargetName())
  throw CTT_Error("Zieldatei  existiert  bereits");
```

10.3.5.7 is_directory

Prototyp

bool is_directory(**const** str& xsPath);

Beschreibung

Gibt *true* zurück, wenn das Betriebssystem meldet dass der Pfad, der durch xsPath bezeichnet wird ein Verzeichnis ist. Andernfalls wird *false* zurückgegeben.

Beispiel:

```
if(is_directory(TargetName()))
  throw CTT_Error("keine korrekte Zieldatei");
```

Anmerkung:

Im Unterschied zur entsprechenden Funktion in boost filesystem, gibt *is_directory* für einen leeren String *false* zurück und produziert keine Ausnahme. Das obige Beispiel ließe sich sonst in der TETRA Arbeitsoberfläche nicht ausführen.

10.3.5.8 file_size

Prototyp

unsigned int file_size(**const** str& xsPath);

Beschreibung

Gibt die Anzahl Bytes zurück, die das Betriebssystem für die Datei meldet. Falls der Pfad *xsPath* nicht existiert, oder falls er ein Verzeichnis bezeichnet, wird eine Ausnahme ausgeworfen.

Beispiel:

```
if(exists(SourceName()) && file_size(SourceName()) == 0)
   throw CTT_Error("die Quelldatei ist leer");
```

10.3.5.9 find_file

Prototyp

bool find_file(**const** str& xsDirectory, **const** str& xsName, str& xsFoundPath, **bool** ci = true);

Beschreibung

Diese Funktion gibt *true* zurück, wenn im Verzeichnis *xsDirectory* eine Datei mit dem Namen *xsName* vorhanden ist. xsFoundPath enthält dann den vollständigen Pfad für diese Datei. Wird die Datei nicht gefunden, gibt die Funktion *false* zurück.
Standardmäßig wird von der Groß-/Kleinschreibung des Namens abgesehen, so wie es unter Windows üblich ist. Optional kann mit dem vierten Parameter die Beachtung der Groß-/ Kleinschreibung für den Namen erzwungen werden.

Beispiel:

```
str sPath;

if(find_file(SourceRoot(), "log.txt", sPath))
{
  ...
```

siehe auch: Unit_dependence Beispiel

10.3.5.10 load_file

Prototyp

bool load_file(str& xs, **const** str& xsFileName)
bool load_file_binary(str& xs, **const** str& xsFileName)

Beschreibung

Öffnet die Datei mit dem Namen *xsFileName* und liest den Inhalt in den String *xs* ein. Wenn dieser Vorgang erfolgreich war wird *true* zurückgegeben, andernfalls *false*.
Wenn der Befehl *load_file* verwendet wird, wird die Datei im Text-Modus geöffnet, d.h. unter Windows, dass alle CR/LF-Kombinationen (Wagenrücklauf/Zeilenvorschub) in ein einzelnes LF-Zeichen übersetzt werden. Dies unterbleibt, wenn die Datei im Binär-Modus mit *load_file_binary* gelesen wird.

Beispiel:

```
str s;
str sFileName = "C:\\Programme\\TextTransformer\\Beispiele\\Atari1\\test.txt";

if( load_file( s, sFileName) )
  out << "file size : " << s.size();
 else
  out << "could not read file: " << sFileName;
```

Vorsicht vor zyklischen Include-Dateien. Wenn zwei Dateien sich unmittelbar oder indirekt gegenseitig einschließen kann es zu einem schwerwiegenden Absturz des TextTransformers kommen. Falls diese Gefahr besteht, sollten sie die geöffneten Dateien mitprotokollieren und eine Datei nur öffnen, wenn sie nicht bereits geöffnet ist.

10.3.5.11 path_separator

path_separator ist eine String-Konstante für den Pfad-Trenner. Die Konstante enthält jeweils den Wert, der vom Betriebssystem verwendet wird, um Verzeichnisse und Dateien in Pfaden voneinander abzutrennen

Windows : "\\"
Unix : "/"

So ergibt z.B.:

```
Pfad = Verzeichnis + path_separator + Dateiname;
```

für sämtliche Betriebssysteme, die von der boost filesystem library abgedeckt sind einen korrekten Pfad

10.3.6 Formatierungsanweisungen

Formatierungsanweisungen werden verwendet, um das Aussehen des generierten Zieltextes zu bestimmen. (Siehe auch Einrückungen)
In den Interpreter ist die Boost Format library von Samuell Krempp integriert, mittels derer Argumente gemäß einem Format-String formatiert werden (ähnlich der Funktionsweise von printf in C).

Die Syntax des Format-Strings und die zu ihm passende Anzahl der Argumente wird vor der Ausführung der Formatierungsanweisung nicht überprüft.

Hinweis: Die folgenden Erklärungen sind im wesentlichen auf die Darstellung der "%|spec|" Spezifikation beschränkt. Zusätzliche Möglichkeiten, die sich enger an die traditionelle printf-Syntax anlehnen, können der Originaldokumentation entnommen werden.

Aus einem Format-String wird ein Objekt erzeugt, dem durch wiederholten Aufruf von operator% Argumente übergeben werden. Jedes dieser Argumente wird in einen String konvertiert, der dann wiederum entsprechend dem Formatstring mit den Strings der anderen Argumente zu einen Gesamtstring kombiniert wird.

```
out << format("writing %|1$|,  x=%|2$| : %|3$|-th try")
                          % "toto" % 40.23 % 50;
// ergibt: "writing toto,  x=40.230 : 50-th try"
```

10.3.6.1 Funktionsweise

1. Wiird *format(s)* aufgerufen, wobei s der Format-String ist, so wird ein Objekt erzeugt, das den Format-String parst, mögliche Direktiven registriert und interne Strukturen für den nächsten Schritt anlegt.

2. Dann werden entweder unmittelbar, wie in

```
out << format("%|2$| %|1$|") % 36 % 77 )
```

oder später, wie in

```
format fmter("%|2$| %|1$|");
fmter % 36; fmter % 77;
```

Variablen in den Formatierer eingespeist. Diese Variablen werden dann entsprechend den im Format-String gegebenen Formatierungsoptionen nach out kopiert und das Format-Objekt speichert das Ergebnis für den nächsten Schritt.

3. Sobald sämtliche Argumente eingeführt wurden, kann das Format-Objekt nach out kopiert werden, oder, man kann auf den gesamten String über die str()-Funktion zugreifen.
Das Ergebnis bleibt im Format-Objekt erhalten, solange bis ihm andere Argumente übergeben werden, wodurch es reinitialisiert wird.

```
// fmter wurde zuvor erzeugt und mit Argumenten gespeist;
// es kann das Ergebnis ausdrucken
out << fmter ;

// Das Ergebnis kann an einen String übergeben werden :
str s  = fmter.str();

// dies ist mehrmals möglich :
s = fmter.str( );

// Es können auch alle Schritte in einem ausgeführt werden :
out << format("%|2$| %|1$|") % 36 % 77;
```

4. Optionnal kann nach Schritt 3 das Format-Objekt wieder benutzt werden, indem bei Schritt 2 neu begonnen wird

```
 fmter % 18 % 39;
```

neue Variablen mit demselben Format-String zu formatieren, wobei die aufwendige Prozedur aus Schritt 1 gespart wird.

Zusammengefasst: die Format-Klasse übersetzt einen Format-String (mit möglicherweise printf-ähnlichen Direktiven) in interne Operationen und gibt schließlich das Ergebnis der Formatierung als einen String zurück, der auch direkt in die Ausgabe geschrieben werden kann.

10.3.6.2 Beispiele

Einfache Ausgabe in neuer Anordnung :

```
out << format("%1% %2% %3% %2% %1% \n") % "11" % "22" % "333";
```

ergibt : `"11 22 333 22 11 \n"`

Einfache Ausgabe ohne Umordnung :

```
out << format("writing %||,  x=%|| : %||-th step \n") % "toto" % 40.23 %
50;
```

ergibt : `"writing toto, x=40.23 : 50-th step \n"`

Präzisere Formatierung durch Positions-Direktiven :

```
out << format("(x,y) = (%|1$+5|,%|2$+5|) \n") % -23 % 35;
```

ergibt : "(x,y) = (-23, +35) \n"

Zwei Arten das Gleiche auszudrucken :

```
out << format("(x,y) = (%|+5|,%|+5|) \n") % -23 % 35;
out << format("(x,y) = (%|1$+5|,%|2$+5|) \n") % -23 % 35;
```

ergibt jedesmal : "(x,y) = (-23, +35) \n"

Neue Formatierungsfunktion : 'absolute Tabellierung', nützlich innerhalb von Schleifen, um sicherzustellen, dass ein Feld in allen Zeilen stets an der gleichen Position gedruckt wird, selbst dann, wenn die Breiten der vorherhehenden Argumente stark variieren können:

```
for(unsigned int i=0; i < names.size(); ++i)
     out << format("%|1$|, %|2$|, %|40t|%|3$|\n") % names[i] % surname[i]
% tel[i];
```

Für drei vstr-Vectoren names, surname und tel, erhält man:

```
Marc-François Michel, Durand,            +33 (0) 123 456 789
Jean, de Lattre de Tassigny,             +33 (0) 987 654 321
```

10.3.6.3 Syntax

format(format-string) % arg1 % arg2 % ... % argN

Der Format-String enthält Text, in dem spezielle Anweisungen durch Strings ersetzt werden, die sich aus der Formatierung der übergebenen Argumente ergeben.

Hinweis: Die Format-Syntax ist angelehnt an die von *printf* der Programmiersprache C. Wer mit *printf* vertraut ist, findet eine Diskussion der Differenzen in Samuel Krempps originaler Dokumentation der Boost Format library bei www.boost.org. Über die standard printf Format-Spezifikationen hinaus, sind neue Funktionen, wie z.B. die zentrierte Ausrichtung implementiert.

In Format-Strings werden Anweisungen akzeptiert, die in mehreren Formen formuliert werden können:

- Traditionelle printf Format-Strings : %spec, wobei spec eine *printf*-Format-Spezifikation ist. Dies wird im TextTransformer als überholt betrachtet.

- %|spec|, wobei spec eine *printf*-Format-Spezifikation ist.Die der Klammerung dienenden senkrechten Striche sollen zum einen die Lesbarkeit des Formatausdrucks erhöhen, vor allem aber machen sie Typ-Konvertierungszeichen überflüssig, die in der ersten Form erforderlich sind. (Optional können sie noch immer verwendet werden.) Z..B. : "%|-5|" wird die nächste Variable auf eine Breite von fünf Zeichen mit Linksausrichtung formatieren (ebenso wie die traditionellen printf-Anweisungen : "%-5g", "%-5f", "%-5s" ..)

- %N%; diese einfache Positionsbezeichnung fordert die Formatierung des N'ten Arguments - ohne jegliche Formatierungsoption.

Eine Spezifikation spec hat die Form : [N$] [flags] [width] [. precision]

Felder innnerhalb von eckigen Klammern sind optional. Jedes dieser Felde wird im folgenden einzeln erklärt :

N $ (optionales Feld) bestimmt, dass die Format-Spezifikation auf das N'te Argument angewendet wird. Falls es nicht vorhanden ist, werden die Argumente nacheinander genommen (und es ist dann ein Fehler, für ein späteres Argument eine Nummer zu verwenden)

flags wird aus einer Reihe der Folgenden gebildet :

Flag	**Bedeutung**
'-'	linke Ausrichtung
'='	zentrierte Ausrichtung
' '	interne Ausrichtung
'+'	Vorzeichen auch für positive Zahlen
'#'	numerische Basis und Dezimalpunkt anzeigen
'0'	mit 0'n auffüllen (nach dem Vorzeichen oder der Basis)
' '	Falls der String nicht mit + oder - beginnt, ein Leerzeichen vor dem konvertierten String einfügen

width bestimmt eine Minimalbreite für einen String, der sich aus einer Konvertierung ergibt. Falls nötig, wird der String durch Ausrichtungs- und Füllzeichen aufgefüllt, die im Format-String angegeben sind (z.B. flags '0', '-', ..)

precision (mit vorangestelltem Punkt), bestimmt die Präzision

Wenn eine Fließkommazahl ausgegeben wird, bestimmte sie die maximale Anzahl der Ziffern:

nach dem Dezimal-Punkt, im wissenschaftlichen Modus
insgesamt, im default Modus

Wenn es für ein String-Argument benutzt wird hat *precision* eine andere Bedeutung : der String nach der durch *precision* gesetzten Anzahl Zeichen abgeschnitten. (Dabei wird ein eventuelles Auffüllen auf eine bestimmte Breite nach dieser Kürzung vorgenommen.)

%{nt} , wobei n eine positive Zahl ist, fügt eine absolute Tabellierung ein. Das bedeutet, dass, falls nötig, der String mit Zeichen aufgefüllt wird, bis seine Länge die Zahl n erreicht hat. (s. Beispiele)

%{nTX} fügt auf gleiche Weise eine Tabellierung ein, benutzt aber X als Füllzeichen, statt des Leerzeichens im default-Zustand.

10.3.6.4 Methoden

unsigned int format::size() const

Die size-Methode kann benutzt werden, um die Anzahl der Zeichen eines formatierten Strings zu ermitteln :

```
format  formatter("%|+5|");
out << formatter % x;
unsigned int n = formatter.size();
```

str format::str() const

Nachdem die Argumente an ein Format-Object übergeben wurden, kann der formatierte String mittels der str-Methode erhalten werden :

```
format  formatter("%|+5|");
formatter % x;
str s = formatter.str();
```

void format::parse(const str& xs)

Löscht die internen Strukturen eines Format-Objekts und parst einen neuen Format-String :

```
format  formatter("%|+5|");
out << formatter % x;
formatter.parse("%|+10|");
```

10.3.7 Sonstige Funktionen

clock_sec	Zeitberechnung
random	Generiert Zufallszahlen
time_stamp	Konvertiert Datum und Uhrzeit in einen String

10.3.7.1 clock_sec

Prototyp

double clock_sec()

Beschreibung

Berechnet die seit dem Programmaufruf verbrauchte "CPU-Zeit" in Sekunden.

clock_sec kann verwendet werden, um die zwischen zwei Ereignissen vergangene Zeit zu bestimmen.

clock_sec ist im Prinzip: std::clock() / CLK_TCK;

Rückgabewert

clock_sec liefert die seit dem Programmaufruf bis jetzt verbrauchte "CPU-Zeit" in Sekunden zurück.

Falls die Prozessorzeit nicht verfügbar ist oder deren Wert nicht dargestellt werden kann, liefert die Funktion -1 zurück

10.3.7.2 time_stamp

Prototyp

str time_stamp()
str time_stamp(const str& xsFormat)

Beschreibung

Konvertiert das augenblickliche Datum und die momentane Uhrzeit in einen String. Wird die Funktion ohne Parameter oder mit einem Leerstring aufgerufen, so enthält der String 26 Zeichen und hat das folgende Format:

```
Mon Nov 21 11:31:54 1983\n
```

Um ein anderes Format zu erhalten, kann der Funktion auch ein Format-String übergeben werden, der aus normalem Text und auf ein "%"-Zeichen folgende Formatierungszeichen zusammengesetzt ist. Die möglichen Formatierungszeichen sind in der folgenden Tabelle aufgelistet:

Formatierungszeichen	Bedeutung	Beispiel
a	Abgekürzter enlischer Wochentagsname	Sun
A	Vollständiger englischer Wochentag	Sunday
b	Abgekürzter englischer Monatsname	Feb
B	Vollständiger englischer englischer Monatsname	February
c	Datum und zeit	Feb 29 14:34:56 1984
d	Tag des Monats	29

H	Sunde im 24-stunden Tag	14
I	Stunde im 12-stunden tag	02
j	Tag im Jahr, ab 001	60
m	Monat im Jahr, ab 01	02
M	Minuten nach der letzten Stunde	34
p	Vor-/Nachmittagsanzeiger (AM/PM)	AM
S	Sekunden nach der letzten Minute	56
U	Sonntagswoche des Jahres, ab 00	
w	Day of the week, with 0 for Sunday	0
W	Montagswoche des Jahres, ab 00	
x	Datum	Feb 29 1984
X	Zeit	14:34:56
y	Jahr im Jahrhundert, ab 00 (veraltet)	84
Y	Jahr	1984
Z	Name der Zeitzone	PST oder PDT

Nur die Datums-und Zeitbestandteile werden ausgegeben, für die Formatspezifizierer vorhanden sind.

Rückgabewert

Ein String mit Datum und Uhrzeit

Beispiele

```
out << time_stamp() << endl;
out << time_stamp("It is %M minutes after %I o'clock (%Z)  %A, %B %d %Y") <<
endl;
out << time_stamp("It is %M minutes after %I o'clock (%Z)") << endl;
out << time_stamp("%A, %B %d %Y") << endl;
```

ergibt zum Zeitpunkt des Schreibens dieses Hilfeabschnitts:

```
Tue Oct 23 00:34:51 2007

It is 34 minutes after 12 o'clock ()  Tuesday, October 23 2007
It is 34 minutes after 12 o'clock ()
Tuesday, October 23 2007
```

10.3.7.3 random

Prototyp

int random(int num)

Beschreibung

Zufallszahlengenerator.

random liefert eine Zufallszahl im Bereich von 0 bis (num-1) zurück. Sowohl num als auch die zurückgelieferte Zufallszahl sind Integerwerte.

Rückgabewert

random liefert einen Zahl zwischen 0 und (num-1) zurück.

10.3.8 Parserklasse-Methoden

Neben den frei definierbaren Klassenmethoden gibt es bereits vorgegebene Klassenmethoden des Parsers.

1. Zugriff aud den Zustand des Parsers
2. Plugin-Methoden

10.3.8.1 Parserzustand

Zu einem bestimmten Zeitpunkt ist der Zustand des Parsprozesses durch die aktuelle Position im Eingabetext und durch die bisher erkannten Token und Regeln gekennzeichnet. Auf einige der Eigenschaften des aktuellen Zustands kann der Interpreter zugreifen. Der Zustand insgesamt ist in TETRA durch die Variable *xState* repräsentiert.

Anmerkung zu den Bezeichnungen ***xState*** und ***State***:

Im Laufe der Entwicklung des TextTransformers wurde statt des Namens ***xState*** auch **State** verwendet. Das vorangestellte 'x' soll zum Ausdruck bringen, dass es sich um eine Parameter-Variable handelt. *State* wurde als Klassenelement benutzt. Mittlerweile ist der Parser-Zustand nur noch als Parameter vorhanden. Deshalb wird jetzt überall die Bezeichnung *xState* verwendet. *State* kann aber weiterhin als Synonym für *xState* verwendet werden. Für den Interpreter macht die Verwendung von State oder xState keinen Unterschied. Siehe auch: xState als Parameter einer Klassenmethode.

Einzelne Eigenschaften des Zustands können über folgende Befehle ermittelt werden:

```
unsigned int size() const
unsigned int length(int sub = 0) const
str str(int sub) const
str str() const
bool matched(int sub) const
bool matched() const
str text(unsigned int from) const
str text(unsigned int from, unsigned int to) const
str copy() const
int itg() const
int itg(int sub) const
double dbl() const
```

```
double dbl(int sub) const

str next_str() const
str next_copy() const
str next_str(int sub) const
unsigned int next_size() const
unsigned int next_length(int sub = 0) const

str lp_str() const
str lp_str(int sub) const
str lp_copy() const
unsigned int lp_length(int sub = 0) const

str la_str() const
str la_copy() const
str la_str(int sub) const
unsigned int la_length(int sub = 0) const

int LastSym() const
unsigned int Line() const
int Col() const
unsigned int Position() const
unsigned int LastPosition() const
unsigned int NextPosition() const
void SetPosition(unsigned int xi );

bool IsSubCall() const
str ProductionName() const
str BranchName() const

int GetState()
void SetState(int xeState);
```

Beispiel:

Quelltext: one two three four
Produktion: "one" "two" "three" "four"

Wenn "two" zuletzt erkannt wurde, gilt:

```
0123456789...
one two three four

xState.str()              : two
xState.copy()             :  two
xState.length()           : 3
xState.size()             : 1
xState.Line()             : 1
xState.Col()              : 8
xState.LastPosition()     : 4
xState.Position()         : 7
```

```
xState.NextPosition()      : 8
```

unsigned int **size**() const

gibt an, wie viele Unterausdrücke an der aktuellen Erkennung beteiligt sind. in diese Zahl eingeschlossen ist die gesamte Erkennung (als Unterausdruck mit dem Index Null).

str **str**(int sub) const

gibt den Textausschnitt an, der von dem Unterausdruck mit dem Index n erkannt wurde. Hierbei steht der Index 0 für die gesamte Erkennung, der Index 1 für den 1. Unterausdruck usw. Wird "str" ohne Parameter aufgerufen, so wird die gesamten Erkennung (= Erkennung mit dem Index 0) zurückgegeben.

bool **matched**(int sub) const
bool **matched**() const

dient zur Abfrage, ob ein bestimmter Unterausdruck zur Gesamtübereinstimmung des regulären Ausdrucks mit dem Text beiträgt.

str **text**(unsigned int from) const
str **text**(unsigned int from, unsigned int to) const

Mit der Funktion *text* erhält man Ausschnitte des Quelltextes.
Wird sie mit nur einem Parameter aufgerufen, liefert sie den Text ab der Position from bis zum Ende des aktuell erkannten Tokens. Mit dem zweiten Parameter kann das Ende des Textabschnitts bestimmt werden.
Falls *from* oder *to* größer sind als die Länge des Quelltextes, oder falls *to* größer ist als *from* wird ein Leerstring zurückgegeben. Falls nur *to* größer ist als die Länge des Quelltextes wird der String von *from* bis zum Ende des Textes zurückgegeben.

str **copy**() const

gibt den String vom Ende des letzten erkannten Tokens bis zum Ende des aktuell erkannten Tokens zurück. xState.copy() ist equivalent zu xState.str(-1) + xState.str():

unsigned int **length**(int sub = 0) const

gibt die Länge des Textausschnitts an, der von dem Unterausdruck mit dem Index n erkannt wurde. Hierbei steht der Index 0 für die gesamte Erkennung, der Index 1 für den 1. Unterausdruck usw. Wird "length" ohne Parameter aufgerufen, so wird die Länge der gesamten Erkennung (= Erkennung mit dem Index 0) zurückgegeben.

int **LastSym**() const

gibt die Nummer des zuletzt erkannten Tokens zurück

unsigned int **Line**() const

gibt die Zeilennummer zurück in der das zuletzt erkannte Token im Quelltext steht. Die Zeilennummerierung beginnt mit 1 in der ersten Zeile.

int **Col**() const

gibt die Spaltennummer zurück in der das zuletzt erkannte Token im Quelltext steht. Die Spaltennummerierung beginnt mit 1 in der ersten Spalte.

unsigned int **LastPosition**() const

gibt die Position des zuletzt erkannten Tokens zurück, d.h. die Anzahl der Zeichen zwischen der Startposition und dem ersten Zeichen des erkannten Tokens.

unsigned int **Position**() const

gibt die Position zurück, an der das zuletzt erkannte Token endet. Diese Position ist um xState.length() größer als xState.LastPosition().

unsigned int **NextPosition**() const

gibt die Position zurück, an der das nächste Token beginnt. Diese Position ist um xState.length() + der Anzahl der auf das letzte erkannte Token folgenden auszulassenden Zeichen größer als xState.LastPosition().

Wurde zuletzt ein SKIP-Token erkannt, so sind Position und NextPosition identisch.
Die Leerzeichen am Ende des durch SKIP abgedeckten Textes können mit *trim_right_copy* entfernt werden.

void **SetPosition**(unsigned int xi);

Mit *SetPosition* kann die aktuelle Position als Anzahl von Zeichen ab Textanfang direkt gesetzt werden. In Folge dieser Methode wird das nächste Token mit dem aktuellen Scanner neu ermittelt. Dies kann z.B. nützlich sein, wenn der Text Metainformationen über die Längen seiner Bestandteile enthält.

bool **IsSubCall**() const

gibt *true* zurück, wenn gerade eine Produktion ausgeführt wird, die vom Interpreter aus aufgerufen wurde. Hier wird eine temporäre Parser-Zustandsvariable *xState* verwendet (i.G. zum Plugin). Innerhalb der Produktionen des Haupt-Parsers gibt diese Funktion *false* zurück.

str **ProductionName**() const

gibt den Namen der aktuellen Produktion zurück.

Der Rückgabetyp ist stets std::string, auch bei der Erzeugung von Unicode-Parsern.

Diese Funktion ist nicht thread-sicher.

str **BranchName**() const

gibt den Namen des letzten Verweigung (Alternative, Option etc.) zurück

Der Rückgabetyp ist stets std::string, auch bei der Erzeugung von Unicode-Parsern.

Diese Funktion ist nicht thread-sicher.

---

int **itg**() const
int **itg**(int sub) const
double **dbl**() const
double **dbl**(int sub) const

Die Funktionen ***itg*** und ***dbl*** konvertieren den Text des zuletzt erkannten Tokens unmittelbar in einen Integer- bzw. einen double-Wert.
Die ***itg***-Funktion konvertiert auch Textabschnitte, die als Oktal- oder Hexadezimalzahlen interpretiert werden können.

---

str **lp_str**() const
str **lp_copy**() const
str **lp_str**(int sub) const
unsigned int **lp_length**(int sub = 0) const

Diese Methode *betreffen* den Textabschnitt, der vom letzten Aufruf einer Produktion erkannt wurde ("lp" steht für "last production").
lp_str gibt diesen Abschnitt ohne die anfänglichen auszulassenden Zeichen zurück, während *lp_copy* den gesamten Abschnitt liefert. Die Methoden *lp_str(int sub)* und *lp_length(int sub)* sind formal äquivalent zu den Methoden *str(int sub)* und *length(int sub)*. Jedoch können sie nur mit den Indizes -1 und 0 aufgerufen werden. Ist der Index == -1 so erhält man die Information über den ausgelassenen Text am Anfang der der Produktion.

Beispiel:

```
Prod1 ::=  Prod3 Prod2 {{cout << xState.lp_copy(); }}
Prod2 ::=  {{cout << xState.lp_copy(); }} Prod1+
Prod3 ::=  ID

Input ::= a b c
Output ::= a b c
```

Prod3 in Prod1 erkennt "a", dass dann in Prod2 ausgegeben wird. Prod2 erkennt dann " b c", was am Ende von Prod1 ausgegeben wird.

---

str **la_str**() const
str **la_copy**() const
str **la_str**(int sub) const
unsigned int **la_length**(int sub = 0) const

Diese Methode *betreffen* den Textabschnitt, der vom letzten Aufruf eines Vorausschau-Parsers erkannt wurde ("la" steht für "look ahead")..
la_str gibt diesen Abschnitt ohne die anfänglichen auszulassenden Zeichen zurück, während *la_copy* den gesamten Abschnitt liefert. Die Methoden *la_str(int sub)* und *la_length(int sub)* sind formal äquivalent zu den Methoden *str(int sub)* und *length(int sub)*. Jedoch können sie nur mit den Indizes -1 und 0 aufgerufen werden. Ist der Index == -1 so erhält man die Information über den ausgelassenen Text am Anfang der Vorausschau.

---

unsigned int **next_size**() const
unsigned int **next_length**(int sub = 0) const
stri **next_str**(int sub) const
str **next_str**() const
str **next_copy**() const

Diese Gruppe von Funktionen sind analog zu den Funktionen, deren Namen nicht mit "next_" beginnen. Sie liefern die entsprechenden Werte für das als nächstes erwartete Token. Im Laufe der Entwicklung des TextTransformers hat sich der Zeitpunkt, zu dem das nächste Token ermittelt wird geändert und es ist nicht völlig auszuschließen, dass sich nochmals Änderungen ergeben könnten. Diese Funktionen sind daher nur unter Vorbehalt zu verwenden.

---

int **GetState**()
void **SetState**(int xeState);

Eine mit einer Reihe von Integerwerten wird der Zustand des Parserzustands gekennzeichnet.

```
typedef enum { epCleared,
               epExpectingToken,
               epExpectingSKIP,
               epExpectingBreak,
               epExpectingEOF,
               epNoProgress,
               epStopped,
               epExpectationError,
               epUnexpectedError,
               epSkipMatchedNeatless,
               epUnknownError,
               epParsedIncomplete,
               epUnknown
             } EPState;
```

Erfahrene Anwender können diese Werte u.U. in *OnParseError* manipulieren, um Fehler auszubügeln.

10.3.8.1.1 Unterausdrücke

In regulären Ausdrücken können die Klammern "(...)" u.a. dazu dienen, die Textteile zu markieren, die von den Unterausdrücken des gesamten regulären Ausdrucks erkannt werden.

Über einen Index-Parameter bei den Funktionen *str* und *length* kann auf die einzelnen Unterausdrücke bzw. ihr Länge zugegriffen werden.
So wird der gesamte erkannte Text durch xState.str() oder xState.str(0) erhalten. xState.str(1) gibt den Textteil zurück, der von dem Unterausdruck mit dem Index 1 erkannt wurde.
Der Index eines Unterausdrucks ergibt sich aus der Nummer seiner öffnenden Klammer innerhalb des gesamten Ausdrucks, wobei die Unterausdrücke beginnend mit 1 von links nach rechts im gesamten Ausdruck gezählt werden.

Der Punkt "Regex Test" im Hilfemenü öffnet einen Dialog zur einfachen Bestimmung der Unterausdrücke eines regulären Ausdrucks.

Folgende Indizes sind definiert:

- 2	der gesamte Eingabetext ab dem Ende des aktuell erkannten Textabschnitts
- 1	die ausgelassenen Zeichen vor Beginn des aktuell erkannten Textabschnitts
0	der aktuell erkannte Textabschnitt
0 < N < size()	den Teil des aktuell erkannten Textabschnitts, der vom N'ten Unterausdruck erkannt wurde
N < -2 or N >= size()	leerer Text

Beispiel:

der Ausdruck:

"(ab)*"

werde auf den Text

"abababab"

angewandt. xState.str(1) enthielte in diesem Fall das letzte "ab" des Textes

Unterausdrücke können auf leere Strings passen, d.h. sie müssen nicht unbedingt an der Übereinstimmung mit einem Text in dem Sinne teilhaben, dass sie einen Teil von diesem abdeckten. Beispielsweise kann der Unterausdruck Teil einer Alternative sein, die nicht zum Text passt.

10.3.8.2 Plugin-Methoden

Die Plugin-Methoden verwenden Daten, die nur für einen Parserdurchlauf gültig sind, oder die während eines Parserdurchlaufs dynamisch veränderbar sind. Insbesondere sind dies

Pfade und Namen der Ein- und Ausgabedateien
Umlenkung der Ausgabe
Einrückungs-Stack
Textbereich-Stack
Dynamische Scanner
Fehlerbehandlung

Die Plugin-Methoden können im Interpreter benutzt werden, wie normale Funktionen.
Nur für den Fall, dass der Parser als C++-Code exportiert werden soll, bestehen für die Plugin-Methoden Besonderheiten.

Ist die const-Option aktiv, so müssen die Methoden als Methoden des Parser-Zustands aufgerufen werden, ist die const-Option hingegen deaktiviert sind beide Aufrufmöglichkeiten gegeben. Z.B.

const ist inaktiv:	*ResetOutput*();
oder:	*xState.ResetOutput*();
const ist aktiv:	*xState.ResetOutput*();

Die Plugin-Methoden sind zu dieser speziellen Gruppe zusammengefasst, um die Erzeugung von mulithread sicherem Parserklassen-Code zu ermöglichen. Die Plugin-Methoden und -Daten befinden sich in einer speziellen Plugin-Klasse, die von der Parserzustandsklasse mit durch die Produktionen "transportiert" werden. Diese Daten lassen sich ohne Einfluss auf den Zustand eines const-Parsers ändern.

10.3.8.2.1 Quelle und Ziel

Prototyp

```
str SourceName()
void SourceName(const str& xsSourceName, bool xbLastFile = true)
str TargetName()
void TargetName(const str& xsTargetName)
str SourceRoot()
void SourceRoot(const str& xsSourceRoot)
str TargetRoot()
void TargetRoot(const str& xsTargetRoot)
```

Beschreibung

Diese Funktionen liefern die aktuellen Quell- und Ziel-Verzeichnisse und -Dateien. Sinnvoll ist die Benutzung dieser Funktionen vor allem für den Transformations-Manager, das Kommandozeilen-Werkzeug und im generierten Code. In der TETRA Arbeitsoberfläche werden

diese Pfade aus dem aktuellen Projekt-Verzeichnis und dem Namen "Unbenannt.txt" zusammengesetzt, wenn nicht zuvor Texte im Quell-Fenster geladen bzw. im Ziel-Fenster gespeichert wurden.

str SourceName()

gibt den Namen der aktuellen Quelldatei inclusive absolutem Pfad zurück.

str TargetName()

gibt den Namen der aktuellen Zieldatei inclusive absolutem Pfad zurück.

str SourceRoot()

gibt den absolutem Pfad des obersten Quellverzeichnisses zurück.

str TargetRoot()

gibt den absolutem Pfad des obersten Zielverzeichnisses zurück.

bool IsLastFile()

gibt an, ob es sich um die letzte einer Reihe von Dateien handelt. Diese Information, kann wichtig sein, wenn eine Reihe von Quelldateien zu nur einer Zieldatei verarbeitet werden, um bestimmte Aktionen mit der letzten Datei abzuschließen. Werden jeweils einzelne Quelldateien in einzelne Zieldateien transformiert, gibt die Funktion *true* zurück.

void SourceName(const str& xsSourceName, xbLast = true)
void TargetName(const str& xsTargetName)
void SourceRoot(const str& xsSourceRoot)
void TargetRoot(const str& xsTargetRoot)

Der Name der aktuellen Quelldatei kann im erzeugten Code nicht automatisch ermittelt werden, kann aber mit dieser Funktion von Programmierer gesetzt werden. Das Gleiche gilt für das Quell- und das Zielverzeichnis.
Beim Setzen des Namens der Quelldatei, kann mit dem zweiten Parameter **xbLast** gesetzt werden, ob es sich bei der Datei um die letzte einer Reihe handelt. Per default ist xbLast == *true*, da in 1:1-Transformationen jede Quelldatei zugleich die letzte der jeweiligen Transformation ist.

str str()

Befindet man sich nicht in der TextTransformer IDE und wurde die Ausgabe nicht in eine Datei umgelenkt, so wird sie in einen Puffer des Plugins geschrieben. Den Inhalt des Plugin-Puffers erhält man mit der str-Methode des Plugins:

```
xState.GetPlugin().str()
```

Beim Aufruf dieser Methode wird der Puffer zugleich geleert.

10.3.8.2.2 Start-Parameter

Parameter, die vor dem Start einer Transformation benötigt werden, können über das Plugin übergeben werden. Je nach Ausführungsweise wird der Parameter-String im Code, in der Kommandozeile, im Transformations-Manager oder in den Projekt-Optionen gesetzt. U.U. kann dieser String mit einem Unter-Parser verarbeitet werden, um ihn in eine Reihe von Parametern zu zerlegen.

str **ConfigParam** ( ) const

Parameter zur Konfiguration einer beliebigen Transformation können mit der Funktion *ConfigParam* gelesen werden, wenn sie vom Anwender zuvor gesetzt wurden.
Ein typisches Beispiel für Konfigurations-Parameter sind Verzeichnisse in denen nach eingeschlossenen Dateien gesucht werden soll.

str **ExtraParam** ( ) const

Parameter zur Konfiguration einer bestimmten Transformation können mit der Funktion *ExtraParam* gelesen werden, wenn sie vom Anwender zuvor gesetzt wurden (s.o.).

10.3.8.2.3 xerces DOM

XML-Dokumente können mit der *DOMDocument*-Klasse des Opensource-Projekts

http://xml.apache.org/xerces-c/

erzeugt, bearbeitet und geschrieben werden. Im *TextTransformer* sind die entsprechenden Operationen in den Interpreter so integriert, dass sich die als *dnode's* verpackten DOM-Elemente fast genauso verhalten wie *node's*. Neben der Lebensdauer der *dnode's* besteht der einzige Unterschied zu den node's darin, dass für die *dnode's* eine Verbindung zur DOMDocument-Klasse hergestellt werden muss. Die Instantiierung der *DOMDocument* Klasse geschieht in der CTT-Xerces-Klasse und das Plugin kann einen Zeiger auf diese Klasse transportieren. Die Verbindung wird durch den einmaligen Aufruf der gleich beschriebenen *GetDocumenttElement*-Funktion für den Wurzelknoten eines dnode-Baum hergestellt.

dnode **GetDocumentElement**();

Um dnode Knoten mit dem DOMDocument des Plugins zu verbinden, muss Code der folgenden Art ausgeführt werden:

```
dnode  root  =  GetDocumentElement();
```

Zu *root* können nun analog zur Beschreibung der Baum-Konstruktion aus *node*-Knoten weitere *dnode*-Knoten hinzugefügt werden.

void **WriteDocument**();
void **WriteDocument**(const str& xsFilename);

Mittels des Befehls *WriteDocument* kann der DOM als XML ausgegeben werden.

```
WriteDocument();
WriteDocument(const  str&  xsFilename);
```

Der Funktion *WriteDocument* kann mit *xsFilename* optional der Name mit vollständigem Pfad für die Datei übergeben werden, in die das Dokument geschrieben werden soll. Ohne diesen Parameter wird das Dokument in *TargetName* geschrieben. *TargetName* darf nicht als Parameter übergeben werden, da sonst versucht würde, die Datei ein zweites mal zu öffnen. Eine existierende Datei mit dem Namen *xsFilename* wird überschrieben.

10.3.8.2.4 Umlenkung der Ausgabe

Syntax

void **RedirectOutput**(const str& xsFilename)
void **RedirectOutputBinary**(const str& xsFilename)
void **RedirectOutput**(const str& xsFilename, bool xbAppend)
void **RedirectOutputBinary**(const str& xsFilename, bool xbAppend)
void **ResetOutput**( )

obsolet:
void **RedirectCout**(const str& xsFileName, bool xbAppend);
void **ResetCout**()

Beschreibung

Nach Aufruf der **RedirectOutput**-Anweisung werden alle Ausgaben in die Datei geschrieben, deren Namen mit dem Parameter *xsFileName angegeben wurde.* D.h. *out* steht dann für die Datei *xsFileName*. Falls eine Datei diesen Namens nicht existiert, wird sie automatisch angelegt. Standardmäßig wird eine vorhandene Datei gleichen Namens überschrieben. Wenn als zweiter Parameter *xbAppend* der boolsche wert *true* übergeben wird, wird die neue Ausgabe an den vorhandenen Text angehängt.

RedirectOutputBinary funktioniet analog wie *RedirectOutput*, jedoch wird die Datei im im Binär-Format geschrieben, d.h. das Zeilenende-Zeichen '\n' wird nicht automatisch in die Kombintion "\r\n" umgewandelt.

Der Befehl wird nur innerhalb der Transformations-Managers, des Kommandozeilenprogramms oder dem exportierten Code wirksam. In der Arbeitsoberfläche wird die Ausgabe stets vollständig in das Ausgabefenster geschrieben.

RedirectOutput kann nicht benutzt werden, wenn für die Zieldatei die UTF8-Kodierung gesetzt ist. Durch diese Option wird RedirectOutput bereits einmal ausgeführt.

Mit **RedirectOutput** kann das Ziel der Transformation auch in mehrere Dateien aufgesplittet werden.

Mit **ResetOutput** wird der ursprüngliche Zustand wiederhergestellt, bei dem die Ausgabe in in die ursprüngliche Datei geschrieben wird.

Den Inhalt des Plugin-Puffers erhält man mit der str-Methode des Plugins:

```
xState.GetPlugin().str().
```

Beim Aufruf dieser Methode wird der Puffer zugleich geleert.

Es ist *weder* nötig **ResetOutput** vor einem Wechsel der Zieldatei mit **RedirectOutput** *noch* am Ende einer Transformation aufzurufen.

10.3.8.2.5 Einrückungs-Stack

Eine häufige Aufgabe bei der Ausgabe von Text besteht darin, bestimmte Abschnitte gegenüber anderen einzurücken. Zur Erleichterung dieser Aufgabe gibt es ein Klassenelement des Plugins, das eine Stapel (einen Stack) von Einrückungs-Werten verwaltet. Ein solcher Wert bezeichnet die Anzahl von Zeichen, z.B. Leerzeichen, die der Ausgabe des eigentlichen Zeilentextes vorausgeschickt werden sollen. Einen Stack solche Werte gibt es, um mit verschachtelte Einrückungen arbeiten zu können.

Beispiel:

Ein typischer Fall sind die Einrückungen, die Programmcode besser lesbar machen sollen:

```
for ( int i = 0; i < 10; i++)
{
  if ( i == 5 )
  {
    func1 ();
    func2 ();
  }
}
```

Hier wird zunächst die for-Schleife eingerückt, dann die if-Abfrage und dann werden die Funktionsaufrufe nochmals weiter eingerückt. Beim Schließen der geschweiften Klammern wird jeweils zu vorherigen Einrückung zurückgegangen.
Befinden sich die nicht eingerückten Texte der Code-Zeilen z.B. in dem Vektor (vstr) v, so kann die obige Ausgabe folgendermaßen erreicht werden:

```
{{
  PushIndent(2);  // gesamten Text 2 Zeichen einrücken

  for( int i == 0; i < v.size(); i++)
  {
    if( v[ I ] == "}";  // auf vorherige Einrückung zurück, wenn schließende
    Klammer folgt
      PopIndent( 2 );
```

```
    out << indent << v[ i ];

    if( v[ I ] == "{";  // weiter einrücken nach öffnender Klammer
      IncrIndent( 2 );
  }
}}
```

Die zentrale Anweisung dieses Beispiels ist:

```
out << indent << v[ i ];
```

indent ist ein Klassenelement, das zum einen den Stack der Einrückungwerte enthält, das zum anderen aber auch - wie hier gezeigt - direkt zur Formatierung der Ausgabe verwendet werden kann, indem es einem Stream übergeben wird, d.h. indem es in die Kette der mit "<<" verbundenen Ausgabeelementen für out eingefügt wird. Normalerweise wird es als erstes der Elemente aufgeführt werden. Andernfalls entstünden statt Einrückungen Lücken im Text.

indent kann direkt nur innerhalb der <<-Kette verwendet werden.

str **IndentStr**() const

Über diese Funktion erhält man einen String, der nur aus Leerzeichen besteht. Die Länge des Strings ist durch den aktuell oben auf dem Stack liegenden Wert bestimmt.

Methoden, die den Wert der Einrückung bestimmen, sind:

void **SetIndenter**(char xc)

Das Zeichen, das zur Einrückung verwendet wird ist standardmäßig das Leerzeichen. Mit der Funktion *SetIndenter* kann auch ein anderes Zeichen hierfür gesetzt werden, z.B. der Tabulator: '\t'.

void **PushIndent**(int xi)

Durch diesen Befehl wird der neue Wert xi zur Einrückung auf den Stack gelegt.

void **IncrIndent**(int xi)

Durch diesen Befehl wird ebenfalls ein neuer Wert zur Einrückung auf den Stack gelegt. Der neue Wert ergibt sich als Summe aus dem letzen Wert und der Zahl xi.

void **PopIndent**()

Der letze Einrückungs-Wert wird vom Stack entfernt.

void **ClearIndents**()

Alle Werte werden aus dem Einrückungs-Stack entfernt.

10.3.8.2.6 Textbereich-Stack

Mittels des Stacks (Stapel) für Textbereiche kann protokolliert werden, welcher Teil des Textes gerade bearbeitet wird. Textteile können z.B. Einleitung, Überschrift, Unterabsatz etc. sein oder in Programmiersprachen der Deklarationsteil einer bestimmten Klasse oder Prozedur bzw. die entsprechenden Definitionsteile.
Die auf der folgenden Seite besprochenen dynamischen Scanner, können sich Token "merken", die zu bestimmten Textbereichen gehören.

Die Methoden des Textbereich-Stack

void **PushScope**(const str& xs)

Mit dieser Methode wird ein neuer Textbereichsname auf den Stack gelegt.

void **PopScope**()

Mit *PopScope* wird der oberste Wert des Stacks entfernt.

void **ClearScopes**()

löscht den gesamten Stack.

str **ScopeStr**() const

ScopeStr gibt den Namen des Textbereichs zurück, der aktuell als oberster auf den Stack gespeichert ist.

10.3.8.2.7 Dynamische Scanner

Dynamische Scanner machen den TextTransformer gewissermaßen lernfähig. Textabschnitte - meist Worte -, die zunächst durch einen allgemeinen regulären Ausdruck erkannt werden, können einem Platzhalter-Token zugeordnet werden. Kommt der gleiche Textteil an späterer Stelle im Eingabetext erneut vor, so kann er nun durch das bisherige Platzhalter-Token erkannt werden.

Da das nächste Token immer schon bereits erkannt ist, kann ein neu hinzugefügtes dynamisches Token erst erkannt werden, wenn das nächste Token konsumiert wurde. Falls nötig, kann auch nach *AddToken* xState.*SetPosition(xState.Position())* aufgerufen werden, wodurch eine Neuerkennung an der aktuellen Position erzwungen wird.

Ab TextTransformer 1.3.4 kann *AddToken* als Übergangsaktion ausgeführt werden. Damit steht es bereits vor Ermittlung des nächsten Tokens zur Verfügung.

Unabhängig von den Projekteinstellungen wird bei Platzhalter-Token stets zwischen Groß- und Kleinschreibung unterschieden.

bool **AddToken**(const str& xsText,
const str& xsDynTokenName)

AddToken fügt dem zuvor zu definierenden Platzhalter-Token *xsDynTokenName* den String *xsText* als Alternative hinzu. Sobald der Parser dann auf das Vorkommen des *xsDynTokenName* genannten dynamischen Tokens im Text prüft und dort der String *xsText* steht, gilt *xsDynTokenName* als erkannt.
Die Funktion gibt bei Erfolg *true* zurück. Wenn das Platzhalter-Token nicht definiert ist, wird *false* zurückgeliefert.

bool **AddToken**(const str& xsText,
const str& xsDynTokenName,
const str& xsScope)

Speziell zum Parsen von Programmiersprachen ist die relativ komplexe *AddToken*-Methode mit drei Parametern konzipiert.
Wird diese aufgerufen, so wird wie bei dem Aufruf mit nur zwei Parametern dem Platzhalter-Token *xsDynTokenName* den String *xsText* als Alternative hinzugefügt. Diese Alternative wird beim anschließenden Parsen aber nur dann im Text erkannt, wenn der aktuelle Textbereich entweder *xsScope* ist oder ein ihm untergeordneter (später hinzugefügter) Textbereich ist.
Wird z.B. die Deklaration einer Klasse geparst, können die Namen der Klassenvariablen dem Scope der Klasse - d.h. ihrem Namen - zugeordnet werden. Im Definitionsteil kann dann erneut der Scope der Klasse gesetzt werden und der dynamische Scanner wird die Namen der Klassenvariablen wiedererkennen. Außerhalb des Scopes können die gleichen Namen eine anderer Bedeutung haben.

Beispiel: " Eval; Eval; Eval "

```
ID
{{
AddToken(xState.str(), "USER_FUNCTION", "FUNCTION_SCOPE");
PushScope("FUNCTION_SCOPE");
PushScope("BLOCK_SCOPE");
}}
// das nächste Token ist bereits erkannt, Eval kann daher frühestens als
übernächstes als USER_FUNCTION erkannt werden.

";"

USER_FUNCTION
// Eval wird als USER_FUNCTION erkannt, weil der aktuelle Scope BLOCK_SCOPE dem
Scope FUNCTION_SCOPE untergeordnet ist

{{
```

```
PopScope();
PopScope();
}}


";"

USER_FUNCTION
// Eval wird nicht erkannt, da aktuell kein Scope gesetzt ist, d.h. auch nicht
der FUNCTION_SCOPE oder ein ihm untergeordneter Scope.
```

Beispiel:

In Grammatiken anderer Parsergeneratoren werden bisweilen die Alternativen einer Syntaxregel so aufgezählt, dass ihnen jeweils der Name der Produktion und das Definitionszeichen vorangestellt wird. Z.B.:

```
Rule1 ::= A B
Rule1 ::= C Rule1

Rule2 ::= D
Rule2 ::= Rule1 E
```

Sollen derartige Regeln in den TextTransformer importiert werden, so bietet sich die Verwendung eines dynamischer Token für die Namen der Regeln an, um die Alternativen in einer Regel zusammenzufassen.

```
{{ str sScope; }}
ID
{{
sScope = xState.str();
AddToken( xState.str(), "SAME", sScope );
}}
"::="
Expression  // GrammarExpression
{{ PushScope(sScope); }}
(
  SAME
  {{ PopScope(); }}
  "::="
  Expression  // GrammarExpression
  {{ PushScope(sScope); }}
)*
```

void **ClearTokens**(const str& xsScope)

Mit dieser Funktion werden alle dynamischen Token gelöscht, die für den Textbereich xsScope definiert wurden. Ist xsScope ein leerer String werden alle Token für alle Bereiche entfernt.

Falls Token bestimmten Textbereichen zugeordnet wurden, sollte vor Beendigung eines Programms *ClearTokens* mit einem leeren String aufgerufen werden, um Speicherlöcher zu vermeiden.

void **UseExcept**(bool xbUseExcept)

Mit dieser Anweisung kann bestimmt werden, ob das Parsen im Fehlerfall durch eine Exception abgebrochen werden soll (xbUseExcept == **true**) oder, ob die Fehlermeldung in einem Container der Plugin-Klasse zwischengespeichert wird und der Abbruch innerhalb des regulären Codes dadurch erfolgt, dass für das nächste Token jeweils die Symbol-Nummer Null zurückgegeben wird.

bool **GetUseExcept**() const

gibt zurück, ob das Parsen im Fehlerfall durch eine Exception unmittelbar abgebrochen werden soll, oder ob der Abbruch über die Rückgabe der Symbol-Nummer Null für das jeweils nächste Token erfolgt.

bool **HasError**() const

HasError gibt *true* zurück, wenn ein Parse-Fehler aufgetreten ist, der keine Exception ausgelöst hat, weil *UseExcept* abgeschaltet wurde. Wenn *HasError true* zurückgibt besteht noch die Möglichkeit unmittelbar eine Aktion auszuführen, bevor das Parsen beendet wird.
Nach der Ausführung einer Produktion, die direkt vom semantischen Code aus aufgerufen wurde, kann mit dem Hauptparser fortgesetzt werden, wenn *UseExcept* abgeschaltet wurde. HasError gibt dann wieder *false* zurück, bis eventuell ein Fehler im Hauptparser auftritt.

void **GenError**(const str& xs)

Durch den Aufruf von *GenError* wird ein Fehler erzeugt, der das Parsen zum Abbruch bringt. Der Funktion wird ein String übergeben, der eine Fehlemeldung enthält. Diese Meldung wird nach dem Abbruch des Parsens in der Log-Fenster angezeigt.
Je nachdem, ob *UseExcept* ein- oder ausgeschaltet ist erfolgt der Abbruch regulär oder durch eine Exception ähnlich wie durch die throw Anweisung. Die durch GenError erzeugte Ausnahmeklasse enthält jedoch zusätzliche Informationen über den Ort, an dem der Fehler auftrat.

void **AddMessage**(const str& xs)
void **AddWarning**(const str& xs)
void **AddError**(const str& xs)

Die Plugin-Klasse enthält nun standardmäßig einen Container (vector) in dem mehrere Meldungen, Warnungen und Fehler gesammelt werden können, um sie erst nach Beendigung des Parsens auszugeben. Hierfür gibt es die drei Methoden *AddMessage*, *AddWarning* und *AddError*, denen jeweils ein Textstring übergeben wird. Diese Strings werden nach Beendigung des Parsens in der Log-Fenster angezeigt oder sie erscheinen im Resultat-Fenster des Transformations-Mangers, wenn das Projekt von dort ausgeführt wurde.

Im Fehlerfall gibt es auch spezielle Ereignisse, die zur Fehlerbehandlung benutzt werden können.

Im exportierten C++-Code erhält man mit den Methoden *MsgBegin* und

MsgEnd zwei Iteratoren, über die die verschiedenen *CTT_Message*-Typen ausgelesen werden können. Mit *HasMessage* kann zuvor geprüft werden, ob der Container nicht leer ist und mit der Methode *GetMsgType* von *CTT_Message* erhält man einen der enumerierten Typen: *eMessage*, *eExit*, *eWarning*, *eError*, *eParseError* oder *eSemError*. Ein Beispiel der verwendung der Iteratoren ist am Ende der main-Schablone zu sehen.

10.3.9 Aufruf einer Produktion

In der Einführung wurde auf die Ähnlichkeit von Produktionen und Funktionen hingewiesen. Der TextTransformer nutzt diese Ähnlichkeit, indem er Produktionen nicht nur innerhalb des zentralen Parser-Gerüstes verwendet, sondern indem er auch die direkte Benutzung von Produktionen als Funktionen erlaubt. Produktionen können

1. als eigenständige Unter-Parser verwendet werden
2. zur Vorausschau dienen.

10.3.9.1 Unter-Parser

Eine Produktion kann direkt aus dem Interpreter-Code aufgerufen werden. Sie gehört dann nicht zur eigentlichen Grammatik des Parsers, in den dieser Interpreter-Code eingebettet ist. Die aufgerufene Produktion ist vielmehr eine Startregel für einen gesonderten Parser und diesem wird ein neuer Eingabetext explizit übergeben. Die Übergabe eines **String-Parameters** unterscheidet den Aufruf eines Unter-Parsers von einem Aufruf zur Vorausschau.

Unter-Parser können z.B. Parameter aus einem Text extrahieren, der von einem Vorausschau-Parser erkannt wurde, bevor mit dem eigentlichen Parsen fortgesetzt wird.

```
IF( Production() ) // Vorausschau-Parser
  {{ Parameters( xState.la_str() ); // Unter-Parser }}
  Production // Haupt-Parser
END
```

Andere Anwendungsbeispiele für Unter-Parser sind die Behandlung von Include-Dateien oder das Einlesen von externen Daten in das Programm.
Soll z.B. die Produktion mit dem Namen *Include* aufgerufen werden, so könnte dies folgendermaßen geschehen.

```
{{
  str buf;
  str sIncludeFile = xState.SourceRoot() + "\\" + xState.str( 1 );

  if( load_file(buf, sIncludeFile ) )
    Include(buf);
   else
     throw CTT_Error( sIncludeFile + " could not be opened");
}}
```

Eventuelle Parameter der *Include*-Produktion werden an den String für den neuen Quelltext angehängt.

Um bereits vor Beginn des Parsens die Anzahl der zu parsenden Zeilen zu ermitteln, kann der Unterparser *CountLines* verwendet werden:

```
CountLines(int& xi) ::=
(
  SKIP EOL
  {{ xi++; }}
)+
```

Aufzurufen wäre *CountLines* in einer Aktion, die ausgeführt wird, bevor das erste Token des Hauptparsers erkannt wurde:

```
{{
int iLines = 0;
CountLines(xState.str(-2), iLines);
}}
```

10.3.9.2 Vorausschau

Produktionen können aus dem Interpreter heraus aufgerufen werden, um zu testen, ob der Eingabetext ab der aktuellen Position zur Produktion passt. In Kombination mit dem IF-Symbol oder WHILE ist es so möglich Texte zu parsen, deren Strukturen sich nicht, oder nur schwer, LL(1)-konform beschreiben lassen.
Bei dieser **Vorausschau** (look ahead) wird die aktuelle Position nicht geändert und der semantische Code der Produktion wird nicht ausgeführt. Eine Parameterübergabe ist daher nicht nötig. Vielmehr erkennt der TextTransformer einen Aufruf zur Vorausschau gerade daran, dass der Produktion **keine Parameter** übergeben werden, im Unterschied zum Aufruf eines Unter-Parsers, der mindestens einen String-Parameter benötigt.
Der Text wird nur soweit getestet bis entweder ein Token erwartet wird, dass in ihm nicht vorkommt oder bis das letzte Symbol der Produktion erkannt wird. So gibt eine Vorausschau einen **bool-Wert zurück**, der *true* ist, wenn der Text zur Produktion passt und *false*, wenn sie nicht passt. Es werden keine Ausnahmen ausgeworfen.
Eine Vorausschau kann selbst von anderen Vorausschau-Produktionen abhängen, die in ihr getestet werden. Die verschiedenen Ebenen der Vorausschau werden als Nummer sowohl in einem gesonderten Feld der Werkzeugleiste als auch als vorangestellte Nummer im Stackfenster dargestellt.

Der von der Vorausschau erkannte Text kann über die Parserzustands-Methode la_str() erhalten werden, z.B. um ihn einem Unter-Parser zu übergeben.

Ein ausführliches Beispiel ist der Java-Parser.

Das Ergebnis der Vorausschau hängt davon ab, ob die Vorausschau-Produktion zu dem System gehört, in dem sei aufgerufen wird oder nicht.

Die Produktion *Ident* wird im gleichen Parser-System verwendet in dem sie auch als Vorausschau-Produktion verwendet wird. Die Eingabe "int" ergibt so das erwartete Resultat: "int gefunden", wenn die Option zum Testen aller Literale aktiviert ist.

```
IF(!Ident())
 "int" {{cout << "int gefunden"; }}
ELSE
 Ident  {{cout << "Fehler"; }}
END

Ident ::= IDENT
IDENT ::= \w+
```

Wird hingegen eine externe Produktion zur Vorausschau verwendet:

```
isIdent ::= IDENT

IF(!isIdent())
 "int" {{cout << "int gefunden"; }}
ELSE
 IDENT  {{cout << "Fehler"; }}
END
```

ergibt die Eingabe "int" den Fehler: "IDENT" erwartet. Da *isIdent* eine Startproduktion ist, die im Hauptparser nicht verwendet wird, kennt sie auch nicht dessen Token. "int" wird in ihr also als *IDENT* interpretiert und !isIdent() ist falsch. Damit wir die ELSE-Alternative gewählt. Im Hauptsystem wurde "int" aber als *int* erkannt, was dann aber von *IDENT* nicht akzeptiert wird.

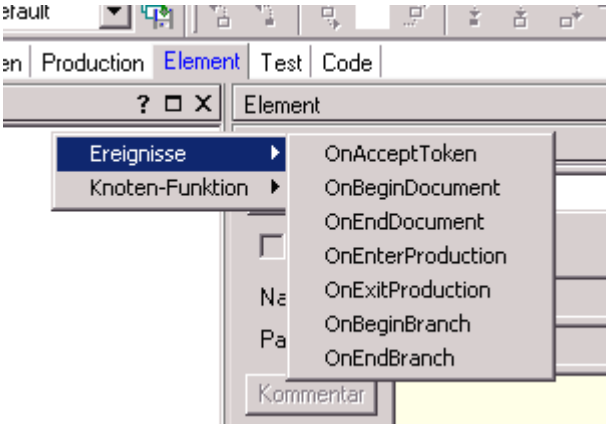
In anderen Fällen ist es von Vorteil eine externe Produktion für die Vorausschau zu verwenden. So wird SKIP im nächsten Beispiel das Satzende-Zeichen finden, auch wenn der Satz mit "Was" oder "Wie" beginnt.

```
isQuestion ::=
SKIP
(
    "?"
  | ( "." | "!" )
     EXIT
)

Sentence ::=
IF(isQuestion())
  Question
ELSE
  NonQuestion
END

Question ::=
(
    "Was" {{ out << "Das solltest du besser wissen als ich!"; }}
  | "Wie"  {{ out << "Ich weiß nicht wie!"; }}
)
```

```
| {{ out << "Ich verstehe deine Frage nicht!"; }}
IDENT+ "?"

NonQuestion ::=
IDENT+ ( "." | "!" )
{{
out << "Das klingt interessant!";
}}
```

10.3.10 Ereignisse

Es gibt eine Reihe von Funktionen, die automatisch immer dann aufgerufen werden, wenn ein bestimmtes Ereignis auftritt. In diesen Funktionen geschieht nichts, solange nicht eine entsprechende Ereignisbehandlung explizit programmiert wurde. Der erste Schritt, um dies zu tun ist es, eine entsprechende Funktion über ein Popup-Menü der Liste der Elementseite einzufügen:



Im Unterschied zu den gleichnamigen Funktionen der TetraComponents werden in diesen Funktionen keine expliziten Parameter übergeben. Wie in allen TETRA-Funktionen gibt es jedoch den impliziten xState-Parameter, über den auf alle Eigenschaften des Parserzustands zugegriffen werden kann.

OnEnterProduction

Das Ereignis *OnEnterProduction* tritt auf, wenn der Parser in eine Produktion verzweigt.

OnExitProduction

Das Ereignis *OnExitProduction* tritt auf, wenn der Parser eine Produktion beendet.

OnAcceptToken

Das Ereignis *OnAcceptToken* tritt auf, wenn ein vom Scanner erkanntes Token in der Grammatik aufgefunden und akzeptiert wird. Im Debugger geschieht dies in dem Moment, wo ein

Terminalknoten verlassen wird.

OnBeginBranch

Das Ereignis *OnBeginBranch* tritt auf, wenn der Parser in eine Option oder in eine Wiederholung verzweigt.

OnEndBranch

Das Ereignis *OnEndBranch* tritt auf, wenn der Parser eine Option oder in eine Wiederholung verlässt.

OnBeginDocument

Das Ereignis *OnBeginDocument* tritt auf, wenn mit dem Parsen eines neuen Quelltexts begonnen wird.

OnEndDocument

Das Ereignis *OnEndDocument* tritt auf, wenn der Parser einen Quelltext beendet.

OnParseError

Das Ereignis *OnParseError* tritt auf, bevor das Parsen mit eine Fehlermeldung abgebrochen wird. Manchmal kann der Abbruch verhindert werden, wenn der Fehler hier ausgebügelt werden kann. Andernfalls besteht in *OnParseError* die Möglichkeit mehr Informationen über die Fehlerumstände auszugeben.

10.4 Testskripte

Ein Testskript ist ähnlich aufgebaut, wie ein Regelskript, jedoch können Testskripts keinen Rückgabewert haben und statt einer optionalen Parameterliste gibt es hier den optionalen Eingabetext. Die Syntax des Skripttextes ist vollständig identisch mit der einer Produktion.

Die Eingabemaske der Regelskripte umfasst folgende Felder:

Name:	eindeutigen Name
Kommentar:	beliebiger Kommentartext
Eingabe:	Eingabetext
Text:	Skripttext des Tests
Erwartete Ausgabe:	Erwartete Ausgabe der Ausführung des Testskripts
Testausgabe	Tatsächliche Ausgabe der Ausführung des Testskripts

Die Angabe von Namen und Text sind erforderlich. Erst wenn diese Felder gefüllt sind, ist es möglich das Skript in die Verwaltung zu übernehmen oder einen Kommentar zu schreiben.

10.4.1 Name

Jedem Test muss ein Name gegeben werden. Der Name kann aus den Buchstaben des Alphabets, Ziffern und Unterstrichen '_' gebildet sein, wobei der Unterstrich nicht an erster Stelle des Namens stehen darf.

Bsple.: test1, int_test, CALC

In das Namensfeld eines neue Tests muss ein von allen Tests unterschiedener Name eingegeben werden. Der Name muss *nicht* von den Namen der Produktions- und Tokenskripte verschieden sein.

10.4.2 Gruppe

Hier kann optional ein beliebiger Gruppenname vergeben werden.
Mehrere Testskripte können durch einen gemeinsamen Gruppennamen zu einer Gruppe zusammengefasst werden, deren Unterregeln gemeinsam kompiliert werden, wenn Zusammenhang testen oder Alle testen ausgeführt wird. Dies bringt eine Zeitersparnis beim Durchführen mehrerer Tests, wenn sie von gleichen Produktionen abhängen.

In seltenen Fällen, kann das Ergebnis eines Tests innerhalb einer Gruppe gegenüber dem isolierten Testen verändert sein. So kann die Menge der möglichen Nachfolger eines SKIP-Symbols, dadurch verändert sein, dass ihm in einem anderen Test ein zusätzliches Token folgt.

10.4.3 Kommentar

Ein Kommentartext zum Test wird in dem gelblich hinterlegten Feld dargestellt.

Dieses Feld wird temporär auch zur Anzeige von Fehlemeldungen etc. verwendet. Um den Kommentar zu ändern muss der Kommentarschalter betätigt werden, der eine gesonderte Dialogbox zum edieren des Kommentartextes öffnet.

10.4.4 Eingabe

In das Eingabefeld kann optional ein Text geschrieben werden, den die Testregel transformieren soll. Der Test verarbeitet diesen Eingabetext auf die gleiche Weise, wie auf der TETRA-Hauptseite eine entsprechende Startregel den Test des Eingabefensters verarbeiten würde.

10.4.5 Code

Die Syntax eines Skripttextes eines Tests ist identisch mit der einer Produktion. Da ein Test normalerweise der Prüfung einer der Produktionen dient, besteht ein Testskript zumeist aus der Deklaration und Initialisierung von Variablen, die der zu testenden Produktion als Parameter

übergeben werden.

10.4.6 Erwartete Ausgabe

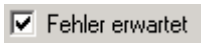
In das Feld: Erwartete Ausgabe eines Tests, wird der Text geschrieben, den die Ausführung des Testskripts produzieren soll. Nach Ausführung des Testskripts wird dieser Text mit der tatsächlichen Ausgabe verglichen, und sollte es Abweichungen geben, wird eine Fehlermeldung erzeugt.

10.4.7 Testausgabe

Das Testausgabefeld ist ein reines Ausgabefeld, d.h. bei Erstellung eines neuen Tests bleibt dieses Feld leer.
Bei der Ausführung des Testskripts wird der generierte Ausgabetext in dieses Feld geschrieben und mit dem der erwarteten Ausgabe verglichen. Sollte es Abweichungen geben wird eine Fehlermeldung erzeugt.

10.4.8 Fehler erwartet

Um erwünschtes Fehlschlagen des Parsens zu testen, gibt es in der Werkzeugleiste der Testseite die Checkbox: Fehler erwartet.



Wird ein Text geparst wird, der nicht regelkonform ist, so sollte der Parser eine Fehlermeldung ausgeben. In diesem Fall bedeutet also gerade das Auftreten des Fehlers, dass der Parser korrekt gearbeitet hat. Ist die Checkbox: "Fehler erwartet" aktiviert, so wird das Auftreten eines Fehlers als richtiges Testergebnis gewertet, d.h. er wird nicht in die Fehlerliste eingetragen. Statt dessen erscheint die Fehlermeldung in dem Fenster für die Testausgabe.

Teil XI

11 Algorithmen

Um effiziente Programme mit dem TextTransformer zu entwickeln und um mögliche Fehler und Konflikte besser zu verstehen ist es hilfreich, die Algorithmen zu kennen, nach denen der TextTransformer bei der Analyse eines Textes vorgeht. Wie in der Einführung erläutert, geschieht die Analyse in zwei Schritten:

Bei der **lexikalischen Analyse** bedarf es eines Algorithmus zur Extraktion des nächsten Tokens aus dem Eingabetext durch einen sogenannten **Scanner**.

Bei der **syntaktischen Analyse** bedarf es eines Algorithmus zur Wahl der nächsten Verzweigung innerhalb des Regelsystems durch den **Parser im engeren Sinne**.

11.1 Scanner-Algorithmus

Ein Scanner versucht das nächste Token zu ermitteln, das auf den aktuellen Text passt. Dabei geht er von einer vordefinierten Gesamtheit aller möglichen Token aus. Im Unterschied zu anderen Parsergeneratoren arbeitet der TextTransformer nicht mit nur einem Scanner sondern es ist auch möglich mit einer Vielzahl kleiner Scanner jeweils nur spezifische Teilmengen der Gesamtheit aller in der Grammatik vorkommenden Token zu testen. Diese Teilmengen bestehen aus den Anfängermengen des aktuellen Knotens und den SKIP-Alternativen. Das Prinzip des Scanners bleibt jedoch das gleiche: für jedes Token des vorgegebenen Reservoirs muss getestet werden, ob es passt oder nicht. Für den Fall, dass es mehrere Token gibt, die passen, existiert ein Algorithmus zur Entscheidung zwischen diesen Kandidaten.

1. Anfängermenge testen

Der TextTransformer versucht zunächst eine Übereinstimmung zwischen dem Text an der aktuellen Position und einem der Token aus der Anfängermenge des aktuellen Knotens zu finden. Gibt es genau ein passendes Token, so ist die lexikalische Analyse an diesem Punkt erfolgreich abgeschlossen.

2. Vorzug der längsten Übereinstimmung

Falls es mehr als eine Übereinstimmung an der gleichen Textposition gibt, so wird dasjenige Token greifen, das die längste Übereinstimmung mit dem Text ergibt, d.h. die größte Anzahl von Zeichen des Eingabetextes abdeckt.

3. Vorzug literaler vor nicht literalen Token

Kommen immer noch mehrere Token in Frage, wird einem literalen Token der Vorzug vor einem nicht literalen gegeben.

4. Vorzug der längsten Übereinstimmung des ersten Unterausdrucks

Falls beide Kandidaten nicht literal sind, wird dasjenige gewählt, dessen erster Unterausdruck die längste Übereinstimmung ergibt.

5. Vorzug der längsten Übereinstimmung des nächsten Unterausdrucks

Bei erneuter Mehrdeutigkeit wird die Länge der Übereinstimmungen der weiteren Unterausdrücke als Auswahlkriterium herangezogen.

6. SKIP-Alternative
Passt keines der Token der Anfängermenge an der aktuellen Textposition, so wird geprüft, ob eine SKIP-Alternative vorhanden ist. Gibt es aus der zum SKIP-Symbol gehörenden Menge genau ein passendes Token, so ist die lexikalische Analyse an diesem Punkt erfolgreich abgeschlossen.

7. nächstliegende Textposition
Gibt es mehrere Kandidaten, zu denen im Text gesprungen werden kann, so wird dasjenige Token ausgewählt, dessen Übereinstimmung mit dem Text der aktuellen Textposition am nächsten liegt.

8. längsten Übereinstimmung (analog Punkt 2-5)
Passen mehrere Token der zum SKIP-Symbol gehörenden Menge an der gleichen Textposition, so wird wie unter Punkt 2-5 gemäß der längsten Übereinstimmung entschieden.

Es ist darauf zu achten, dass bei Token, die in einer Verzweigung alternativ zueinander stehen, nicht eine Definition die Definition eines anderen Tokens umschließt. TETRA kann für diesen Fall keinen Warnhinweis geben.

Beispiel:

```
IDENT = \w+
NUMBER = \d+
```

Da "\w" u.a. Menge der Ziffern enthält, umschließt die Definition von IDENT auch die von NUMBER. Im Falle einer Alternative

```
IDENT | NUMBER
```

ist nicht auszumachen, welches der beiden Token greift, wenn eine reine Zahl in der Eingabe steht (->Anmerkung). Die Alternative von IDENT und NUMBER muss nicht explizit in einer Regel formuliert sein um zu Problemen führen zu können, sondern kann in der Anfängermenge von Verzweigungen versteckt sein, von denen aus IDENT und NUMBER indirekt erreichbar sind.
Allgemein gilt, dass es empfehlenswert ist ein Token möglichst spezifisch zu definieren. So würde z.B.

```
IDENT = [[:alpha:]|_]\w*
```

das obige Problem von vornherein ausschließen, da IDENT dann nicht mit einer Ziffer beginnen kann, NUMBER aber immer mit einer Ziffer beginnen muss.

Anmerkung: Tatsächlich greift IDENT, während bei einer Definition von IDENT als "[[:alpha:]|_|\d]+" NUMBER greift. Dies wird durch die interne Implementation der Regex-Library bestimmt und kann z.Z. nicht allgemein in Regeln gefasst werden.

11.2 Parser-Algorithmus

Die Aufgabe des Parsers besteht in der syntaktischen Analyse von Texten. Er geht aus von einem Regelsystem (**Grammatik**), das die erlaubten Möglichkeiten der Abfolgen von Token definiert. Bei jedem neuen Schritt des Parsers, testet dieser, ob das aktuell ermittelte Token ein erlaubtes Folge-Token ist oder nicht. Falls nicht, ist das Parsen des Textes gescheitert, falls ja, wird zu der durch das Token bestimmten Alternative des Regelsystems fortgeschritten.
Die Entscheidung über die Zulässigkeit des nächsten Tokens wird anhand der Zugehörigkeit zu vorberechneten Tokenmengen gefällt: den Anfängermengen der möglichen Nachfolgerknoten samt deren SKIP-Alternativen. Hier ist die Funktion des Parsers mit den Scannern verschränkt, denn diese Tokenmengen bilden genau die Gesamtmengen der einzelnen Scanner, von denen im vorherigen Abschnitt die Rede war. In den Projekteinstellungen kann die Menge der Token, die an einer Stelle des Regelsystems getestet wird auf die Menge der für den Fortgang des Parsens erlaubten Token eingeschränkt werden. Dies ergibt einen Geschwindigkeitsvorteil insbesondere dann, wenn das Lexikon des Parsers groß ist.

Der Parser-Algorithmus ist nun also dieser:

1. Anfänger der Startregel testen

Zu Beginn der Textanalyse ist noch kein Token erkannt. Daher wird durch einen lokalen Scanner der Textanfang mit der Menge von Token verglichen, die als Anfänger der Startregel in Frage kommen. (Sollen auch andere Produktionen des Parser unmittelbar aufgerufen werden können, so müssen entsprechend auch für diese die Mengen der terminalen Anfänger berechnet und ein Scanner konstruiert werden.)

2. Fortgang zum erwarteten Terminalsymbol

Ist ein Token gefunden, so wird zu demjenigen nachfolgenden Knoten fortgegangen, zu dessen Anfängermenge das Token gehört. Dies wird durch die Reihe der löschbaren Knoten solange wiederholt, bis der Knoten erreicht ist, der das Terminalsymbol repräsentiert.

3. Nachfolger innerhalb und außerhalb der Produktion testen

Jedem Terminal-Knoten ist ein Scanner zugeordnet, der aus dem Bereich der möglichen nachfolgenden Terminalsymbole dasjenige ermittelt, das zum aktuellen Text passt. Nun müssen allerdings zwei Fälle unterschieden werden:

a) innerhalb der dem Terminalsymbol übergeordneten Produktion liegt zwischen diesem Terminal und den die Produktion abschließenden Symbolen stets ein weiteres Terminalsymbol. In diesem Fall wird das nächste Token ermittelt und es wird fortgesetzt wie unter Punkt 2.

b) dem Terminalsymbol folgt in seiner Produktion eine löschbare Struktur derart, dass deren letztes Symbol die Produktion abschließt. In diesem Fall hängt die Menge der möglichen Nachfolger des Terminalsymbols von der Menge der Nachfolger der Produktion ab. Eine Produktion kann aber an verschiedenen Stellen einer Grammatik aufgerufen werden, und an den jeweiligen Stellen können jeweils andere Token nachfolgen. Es wird also zunächst die unvollständige Menge derjenigen Nachfolger getestet, die innerhalb der aktuellen Produktion erwartet werden, und weiter wird mit Punkt 3 fortgesetzt.

Bemerkung:
Der TextTransformer ist ein interpretierter Parsergenerator. Er ist ein Parser, der Regelskripte parst um aus ihnen einen neuen Parser zu erzeugen. Der erzeugte Parser wiederum parst Eingabetext, um ihn zu transformieren. Da der Parser des TextTransformers von ihm selbst erzeugt ist, unterscheiden sich die Algorithmen beim Parsen der Skripte und beim Parsen des Eingabetextes nicht.

11.3 Tokenmengen

Die Entscheidung über die nächste Verzweigung in der Grammatik kann in speziellen Fällen von einer Vorausschau im Text oder vom semantischen Prädikaten abhängig gemacht werden. Im Regelfall aber, um den es im Folgenden geht wird diejenige grammatische Alternative gewählt, deren Anfängersymbol im Text als nächstes erkannt wird. Neben den bereits erläuterten Präferenzregeln kann dies von der Menge der Token abhängen, welche getestet werden: Token, die nicht getestet werden, können auch keine Konflikte verursachen. Deshalb gibt es in den Projekt- und in den lokalen Einstellungen die Option, nur erwartete Token zu testen. Es kann aber auch gerade erwünscht sein, Konflikte frühzeitig zu erkennen. So dürfen z.B reservierte Worte einer Programmiersprache nicht als Namen für Variablen verwendet werden:

```
double int;  // Fehler
```

In diesem Fall ist die Option nur erwartete Token zu testen zu deaktivieren. Zu erörtern bleiben aber die Tokenmengen bei den Verwendung von Produktionen außerhalb der Hauptparsers.

Tokenmengen in Einschlüssen, Unterparsern und bei der Vorausschau.

Besonders im Falle von Kommentaren ist augenfällig, dass hier Konflikte mit den Token des Hauptparsers unerwünscht sind.

```
CppComment ::= "/*" ( SKIP | STRING )* "*/"
// ! diese Definition ist für verschachtelte Kommentare nicht geeignet

/* int iCount : Zähler */
```

Würde das Schlüsselwort *int* hier erkannt, könnte der Kommentar nicht geparst werden.
Daher können Einschlüsse im TextTransformer ein neues Produktionen-System bilden, das von der Tokenmenge des Hauptparsers unabhängig ist.
Für Vorausschau-Produktionen gilt genau das Gegenteil: hier ist es zumeist erwünscht, das die gleichen Token erkannt werden wie im Hauptparser.

Folgende **Regeln** ermöglichen eine flexible Anpassung der Tokenmengen an die jeweiligen Zwecke:

1. Jede Produktion, die nicht von einer anderen Produktion aufgerufen wird, d.h. jede Start-Produktion bildet das Fundament für ein eigenständiges Produktionen-**System** mit eigener Tokenmenge. Diese Menge ergibt sich als Vereinigung aller in dem System vorkommenden Token. (In dem System verwendete Vorausschau-Produktionen gelten hier nicht als *aufgerufen*, gehören also nicht automatisch zum System). Die Startregel des Hauptparsers bildet das erste System.

2. Wird eine Produktion in mehreren Systemen verwendet, dann werden die Tokenmengen dieser Systeme vereinigt.

Beispiel:

```
Prod1  ::= IF( Prod2() ) Prod2 ELSE Prod3 END
Prod2  ::= "a" "b"
Prod3  ::= IF( !Prod4() ) "c" "d" ELSE  ID+ END
Prod4  ::= SKIP "h"
```

Ausgehend von der Startregel *Prod1* sind die Regeln *Prod2* und *Prod3* zu erreichen. Die Tokenmenge dieses Systems ist: "a", "b", "c", "d", ID.

Prod2 wird als Vorausschau verwendet. Da diese Produktion aber auch zum System der Startregel gehört, ist die in *Prod2* von globalen Scannern getestete Tokenmenge mit der des Hauptsystems identisch: "a", "b", "c", "d", ID.

Die Vorausschau mit *Prod4* hingegen ist vom Hauptsystem unabhängig. *Prod4* bildet somit ein eigenes System, deren Tokenmenge nur aus "h" besteht.

Würde nun *Prod4* erweitert zu:

```
Prod4  ::= SKIP "h" Prod2?
```

dann wäre *Prod2* ein Teil von beiden bisherigen Systemen. In diesem Fall werden gemäß dem obigen Punkt2 die Tokenmengen der beiden Systeme vereinigt: "a", "b", "c", "d", "h", ID.

Diese Erweiterung von *Prod4* kann zu Folge haben, dass "a" im Text "a h" nicht mehr von *SKIP* übersprungen wird.

TextTransformer

Teil XII

12 Grammatiktests

Ein TETRA-Programm muss bestimmten (für eine sogenannte LL(1)-Analyse erforderlichen) Bedingungen genügen, die im folgenden erklärt werden. Wenn eine der Bedingungen nicht erfüllt ist, erzeugt TETRA **Fehlermeldungen** oder **Warnhinweise**. Im Falle einer Fehlermeldung muss der Fehler behoben werden, bevor das Programm ausgeführt werden oder als Code exportiert werden kann. Im Falle von Warnhinweisen kann das Programm zwar ausgeführt und exportiert werden, es enthält jedoch Mehrdeutigkeiten, die von TETRA aufgelöst werden, indem es automatisch eine der Möglichkeiten auswählt. Generell wählt TETRA bei Mehrdeutigkeiten die bei der Syntaxanalyse zuerst angetroffene Alternative. In seltenen Fällen kann diese Auflösung den Intentionen des Programmautors widersprechen.

Im einzelnen muss eine TETRA-Grammatik folgenden Bedingungen genügen:

1. Vollständigkeit: Jedes Symbol (Nonterminal) ist durch genau eine Regel definiert und es darf kein Nonterminal geben, das nicht definiert wurde.

2. Ableitbarkeit: Jede Regel muss von der Startregel aus erreichbar sein.

3. Zirkelfreiheit: Kein Nonterminal darf direkt oder indirekt von sich selbst abgeleitet sein.

4. Terminalisierbarkeit: Jedes Nonterminalsymbol muss in Terminalsymbole abgeleitet werden können.

5. Eindeutigkeit: Die Grammatik muss LL(1) sein.

12.1 Vollständigkeit

Jedes Nonterminal ist durch genau eine Regel definiert und es darf kein Nonterminal geben, das nicht definiert wurde.

Der erste Teil dieser Bedingung ist in TETRA automatisch gewährleistet, da es in der Verwaltung nicht möglich ist zwei Produktionen mit dem gleichen Namen einzufügen.

Der zweite Teil hingegen muss von TETRA gesondert geprüft werden. Es kann vorkommen, dass bei der Definition einer Regel ein nicht existierender Symbolname verwendet wird. Das Syntaxhighlighting hilft bereits einen derartigen Fehler zu vermeiden: dieser Name wird nicht in Fettschrift dargestellt. Die Verwendung des nicht definierten Namens wird von TETRA bereits während des Parsens bemerkt und mit einer Fehlermeldung quittiert:

```
Unbekanntes Symbol: "xxx".
.
```

12.2 Ableitbarkeit

LL(1)-Parser wie TETRA müssen normalerweise die Bedingung erfüllen, dass jede Produktion von der Startregel aus erreichbar ist.

In TETRA ist dieses Prinzip allerdings nur vermittelt gültig. TETRA's Familienkonzept erlaubt ausdrücklich das Vorhandensein von Produktionen in der TETRA-Verwaltung, die nicht sämtlich einer Startregel untergeordnet sind. Während der Ausführung eines TETRA-Programms oder der Codegenerierung ist diese Bedingung dennoch automatisch garantiert, da TETRA genau das Ensemble von Regeln zusammenstellt, die von einer momentanen Startregel ableitbar sind.

12.3 Terminalisierbarkeit

Jedes Nonterminalsymbol muss in Terminalsymbole abgeleitet werden können. Die folgende Produktion wäre ein Beispiel für eine Regel, die dieses Prinzip verletzt:

```
X ::= "(" X ")"
```

Das Parsen dieser Regel ergibt folgende Fehlermeldung

Fehler: "X" kann nicht in Terminalsymbole abgeleitet werden

12.4 Zirkularitätstest

Kein Nonterminal darf direkt oder indirekt von sich selbst abgeleitet sein. Die folgende Produktion wäre ein Beispiel für eine Regel, die dieses Prinzip verletzt:

```
Circular1 ::= Circular2 | "T"
Circular2 ::= Circular1 | "T"
```

Das Parsen dieser Regeln ergibt die Fehlermeldungen:

Zirkuläre Ableitung: "Circular1" . "Circular2"
Zirkuläre Ableitung: "Circular2" . "Circular1"

Anmerkung:

Folgende Regeln ergeben hingegen die Fehlermeldungen, dass sie nicht terminalisierbar seien.

```
Circular1 ::= Circular2
Circular2 ::= Circular1
```

Der Terminalisierbarkeitstest wird vor dem auf Zirkularität ausgeführt und beim Auffinden einer Zirkularität wird die weitere Analyse abgebrochen.

12.5 LL(1)-Test

Die Grammatik muss LL(1) sein. Das bedeutet, dass der Parser stets in der Lage sein muss, im Vorblick auf das nächste im Text identifizierte Symbol zu entscheiden, welche der möglichen Grammatik-Alternativen zu wählen ist. Die LL(1)-Analyse arbeitet sehr schnell und ihr Prinzip ist einfach zu verstehen. Jedoch muss bei der Formulierung der Produktionen stets auf die Einhaltung der Bedingung geachtet werden.

Beispielsweise genügt die folgende Produktion (für eine Anweisung) nicht der LL(1)-Bedingung:

```
ident ":=" Expression
| ident ( "(" ExpressionList ")" )?
```

Beide Alternativen beginnen mit dem *ident* Symbol und, wenn der Parser zu dieser Produktion gelangt und *ident* als nächstes Token erkennt, kann er die Alternativen nicht unterscheiden. (Dazu müsste er ein weiteres Symbol vorausschauen, was er aber nicht kann.) Die Produktion kann jedoch leicht so umgeformt werden, dass alle Alternativen mit unterschiedlichen Token beginnen:

```
ident
(
  ":=" Expression
  |  ( "(" ExpressionList ")"  )?
 )
```

Es gibt LL(1)-Konflikte, die nicht so einfach aufzufinden sind, wie in obigem Beispiel. Für den Programmierer kann es sehr schwer sein, sie zu finden, wenn er kein Werkzeug benutzt, dass die Grammatik prüft. Das Ergebnis wäre ein Parser, der in einigen Situationen die falsche Alternative wählt. Der TextTransformer prüft, ob die Grammatik der LL(1)-Bedingung genügt und gibt Hinweise wo Verletzungen vorliegen und zu korrigieren sind.

Wenn LL(1)-Konflikte nicht anders zu lösen sind, können IF...ELSE...END oder WHILE...END Strukturen verwendet werden.

12.6 Warnungen

Während die oben genannten Fehler einer TETRA-Grammatik ihren Gebrauch unmöglich macht, gibt es eine Reihe von Mehrdeutigkeiten einer Grammatik, bei denen TETRA automatisch eine bestimmte Alternative wählt und lediglich eine Warnmeldung erzeugt.
Die Warnungen können auf Fehler hinweisen, in den meisten Fällen jedoch wird TETRA genau das Programm erzeugen, das dem Programmautor vorgeschwebt haben dürfte.

Folgende Warnungen können Auftreten:

1. "X" ist löschbar

2. LL(1) Fehler: "X" ist Anfänger mehrerer Alternativen
3. LL(1) Fehler: "X" ist Anfänger und Nachfolger löschbarer Strukturen

.

12.7 Löschbarkeit

Die Warnmeldung

"X" ist löschbar

erscheint, wenn die Produktion "X" auf einen leeren Text passt. Ist beispielsweise "X" definiert durch:

```
X = "la" *
```

so kann durch die Regel ein Text wie "la la la" erkannt werden, aber auch der leere Text "" passt zur Regel, da die "*"-Wiederholung auch die null-maligen umfasst. Die Produktion X ::= "la" + hingegen ist nicht löschbar.

Diese Warnmeldung kann ein Hinweis darauf sein, dass unabsichtlich eine löschbare Alternative erzeugt wurde.

Produktionen, die nur aus semantischen Aktionen bestehen sind im Prinzip auch löschbar. Eine Warnmeldung wird hier aber nicht erzeugt, da diese Regeln nicht zu fehlerhaften Erkennungen führen können, aber sinnvoll als Hilfs-Funktionen eingesetzt werden können.

Die Erzeugung dieser Warnung kann durch Setzen der entsprechenden Projekt-Option unterdrückt werden.

12.8 Anfänger mehrerer Alternativen

Die Warnmeldung

LL(1) Fehler: "X" ist Anfänger mehrerer Alternativen

erscheint, wenn Alternativen mit demselben Terminalsymbol beginnen. Z.B. in der folgenden Regel:

```
Gruss ::= "guten" "morgen" | "hallo" | "guten" "tag"
```

beginnen sowohl der erste als auch der letzte Gruß mit dem Wort "guten". Sobald TETRA im Eingabetext dies Wort findet, würde die erste der Alternativen gewählt, auch, wenn das Wort "tag" folgt. Die Regel sollte folgendermaßen umgeschrieben werden.

```
Gruss ::= "guten" ("morgen" | "tag" ) | "hallo"
```

12.9 Anfänger und Nachfolger löschbarer Strukturen

Die Warnmeldung

LL(1) Warnung: "X" ist Anfänger und Nachfolger löschbarer Strukturen

erscheint, wenn mit dem Terminalsymbol "X" eine löschbare Struktur beginnt und das gleiche Terminalsymbol auf diese Struktur folgen kann. Löschbare Strukturen sind (...)? und (...)*. Ein Beispiel wäre die folgende Regel zur Erkennung einer Zahl, die möglicherweise Nachkommastellen aufweist:

```
Zahl ::= ( Ziffernfolge "," )? Ziffernfolge
// LL(1) Warnung: Ziffernfolge ist Anfänger und Nachfolger löschbarer Strukturen
```

Trifft nun ein TETRA-Programm auf eine Ziffernfolge, so wird automatisch die löschbare Struktur getestet. Folgt dann kein Komma, wird das Programm mit einer Fehlermeldung abgebrochen. Die Regel sollte daher besser umgekehrt formuliert werden:

```
Zahl ::= Ziffernfolge ( "," Ziffernfolge )?
```

Da von der gewählten Startregel abhängt welche Produktionen geparst werden, kann von dieser Wahl auch abhängen, ob ein Token Anfänger und Nachfolger einer löschbaren Struktur ist oder nicht.

Ein Beispiel für diesen Satz erhält man, wenn man das letzte Beispiel weiter ausspinnt. So wäre denkbar, dass die Zahl-Produktion in einer Zahlenpaar-Produktion steht, in der zwei Zahlen durch Kommas getrennt werden sollen. Z.B.:

```
Zahlenpaar  ::= Zahl "," Ziffernfolge
```

Erneut ist das Komma Anfänger und Nachfolger einer löschbaren Struktur.

Die Erzeugung dieser Warnung kann durch Setzen der entsprechenden Projekt-Option unterdrückt werden.

Ein weiteres Beispiel ist in der Literatur ist als ***dangling else*** bekannt. Folgende zwei Regeln verletzen die LL(1)-Bedingung:

```
Statement ::= IfStatement | ...
IfStatement ::= "if" Expression "then" Statement
                  ( "else" Statement )?
```

Die Anweisung:

```
if a then if b then c else d
```

würde TETRA folgendermaßen interpretieren

```
if a then
```

```
{
   if b then
      c
      else
      d
}
```

d.h. TETRA würde den *else*-Zweig dem *if b* zuordnen. Logisch betrachtet wäre jedoch folgenden Zuordnung ebenso möglich:

```
if a then
{
   if b then
      c
}
else
d
```

12.10 Konkurrierende SKIP-Knoten

Die Fehlermeldung

"X" ist ein SKIP-Knoten mit benachbarten Skip-Knoten

wird erzeugt, wenn in einer Alternative mehr als ein SKIP-Knoten enthalten ist.

In TETRA gibt es bisher keinen Algorithmus zur eindeutigen Behandlung solcher Fälle.

Beispiel:

```
Regel1 ::=
(
"TETRA"
| SKIP
)*

Regel2 ::=
(
"Texttransformer"
| SKIP
)*

Regel3 ::= Regel1 | Regel2
```

Eine einfache Umformulierung beseitigt hier das Problem:

```
Regel3 :: =
(
"TETRA"
|"Texttransformer
```

```
| SKIP
)*
```

Schwieriger wird es, wenn die benachbarten SKIP-Knoten in verschiedenartigen Strukturen eingebettet sind.

12.11 Verschiedene SKIP Nachfolger

Enter topic text here.

12.12 Verschiedene ANY Nachfolger

Enter topic text here.

12.13 Linksrekursion

Ein weiterer möglicher Grammatikfehler ist die Linksrekursion. Die folgende Regel ist das einfachste Beispiel einer Linksrekursion:

a = a | B

Die Prüfung, ob diese Regel auf einen Eingabetext passt führt zu einem unendlichen Regress. Um die Regel **a** zu testen, muss bevor die Alternative **b** geprüft werden kann **a** nochmals getestet werden, usw. Linksrekursionen sind demnach in TETRA nicht zulässig, können aber nicht durch einen Grammatiktest erkannt werden.

Glücklicherweise lassen sich Linksrekursionen vermeiden. Die linksrekursive Regel:

a = a C | B

kann umgeschrieben werden. *a* muss gewiss mit *B* beginnen. Auf *B* kann *C* folgen und auf *B C* kann ein weiteres *C* folgen. Das formale Resultat ist also:

a = B C *

Angewandt auf das erste Beispiel wäre C ein leeres Symbol und für a gilt:

a = B

Anmerkung:
Produktionen für den weit verbreiteten Parsergenerator **Yacc** sind häufig linksrekursiv. Yacc hat damit kein Problem, da er nicht wie TETRA ein Topdown-, sondern einen Bottomup-Parser ist. (Yacc kommt dafür nicht mit Rechtsrekursionen zurecht.) Will man eine für Yacc geschriebene Grammatik in eine TETRA-Grammatik übersetzen, so hilft die eben genannte Regel weiter.

12.14 Zirkuläre Vorausschau

Eine Vorausschau kann nicht ausgeführt werden, wenn sie zirkulär ist.

Offensichtlich zirkulär wäre z.B. folgende Vorausschau:

```
expression  ::=
IF( expression() )
...
```

In der Vorausschau wird erneut auf *expression* getestet und in diesem Test wiederum usw.

Die Zirkularität kann aber auch versteckt sein, wie in den folgenden Produktionen:

```
expression  ::=
IF( factor() )
...

factor  ::=
IF( expression() )
...
```

TETRA versucht derartige Zirkularitäten zu erkennen und produziert entsprechende Fehlemeldungen. Als zusätzliche Sicherheitsmaßnahme wird der Stack für die Vorausschau-Produktionen begrenzt.

TextTransformer

Teil XIII

13 Codeerzeugung

Die Professional-Version des TextTransformers ist dadurch ausgezeichnet, dass es hier möglich ist, den Quellcode für eine C++-Klasse des gebildeten Parsers zu exportieren. Der TextTransformers erzeugt sämtlichen Code, der für ein eigenständiges ausführbares Programm erforderlich ist. Ebenso lässt sich der generierte Code als Teil in größere Anwendungen einbinden.

Die Verwendung des generierten Codes hat drei Vorteile gegenüber der Verwendung eines Projekts im TextTransformer selbst:

- das mit dem Code erzeugte Programm ist unabhängig vom TextTransformer zu benutzen
- das Programm ist schneller
- die semantischen Aktionen sind nicht auf die Untermenge der in TETRA interpretierbaren C++-Anweisungen beschränkt

In der Einleitung wurde dargestellt, dass eine Produktion als Spezifikation zur Erzeugung eine Funktion angesehen werden kann. Die Codeerzeugung ist nun die Anwendung dieser Spezifikation, d.h. für jede Produktion wird eine entsprechende Funktion erzeugt.

Der C++-Code für die semantischen Aktionen wird dabei aus den entsprechenden Abschnitten der einzelnen Produktionen entsprechend der Projekteinstellung durch bloßes Kopieren oder durch Rekonstruktion übernommen. Der C++-Code für den eigentlichen Parser wird von TETRA an gekennzeichneten Positionen in Quellcode-Schablonen geschrieben.

Zur Kompilierung eines von TETRA erzeugten Parsers ist neben der Bibliothek für reguläre Ausdrücke von Dr. Maddock noch einiger unterstützender C++-Quellcode erforderlich.

13.1 Code-Schablonen

Der C++-Code für den eigentlichen Parser wird von TETRA an gekennzeichneten Positionen in Quellcode-Schablonen geschrieben.
Der Name der generierten Parserklasse wird dabei aus dem Namen der Startregel abgeleitet.
Es gibt

1. eine Schablone für den Header
2. eine Schablone für die Implementierung
3. eine Schablone für den Aufruf des Parsers

Die vorgegebenen Standardschablonen können spezifisch an das Projekt angepasst werden.

Zusätzlich wird eine Datei mit dem Namen *jamfile.txt* generiert, die als Grundlage für ein "jamfile" zur Verwendung mit boost bjam dienen kann.

13.1.1 Name der Parserklasse

Der Namen der Produktion, die als Startregel ausgewählt ist, wird zur Grundlage des Namens der Klasse, die bei der Generierung von Source-Code für Ihren Parser angelegt wird.

Dem Namen der Startregel wird ein großes 'C' vorangestellt und das Wort "Parser" angefügt.

Beispiel: Ist der Name der Startregel: "Pascal", so ist der Name der Parserklasse:

"CPascalParser":

Anm.: das 'C' steht für "class". Die Voranstellung dieses Buchstabens entspricht der Namenskonvention von Visual C++.

13.1.2 Header-Schablone

Die unmodifizierte Schablone für den Header einer TETRA-Parserklasse ist folgende:

```
//-------------------------------------------------------------
//   ttparser_h.frm
//   TextTransformer C++ Support Frame
//   Author: Dr. Detlef Meyer-Eltz
//   http://www.texttransformer.de
//   http://www.texttransformer.com
//   Meyer-Eltz@t-online.de
//
//   March, 2006  Version 1.1.0
//-------------------------------------------------------------


#ifndef -->ParserHeaderSentinel
#define -->ParserHeaderSentinel


#ifndef tt_parserH
#include "tt_parser.h"
#endif

#ifndef tt_symbolentryH
#include  "tt_symbolentry.h"
#endif

// the following includes and the according typedefs can be removed,
// if you don't use them

#include "boost/format.hpp"

#ifndef tt_mapH
```

```
#include "tt_map.h"
#endif

#ifndef tt_vectorH
#include "tt_vector.h"
#endif

#ifndef tt_nodeH
#include "tt_node.h"
#endif

namespace tetra
{

class -->ParserClassName : public CTT_Parser< -->CharType, -->PluginType >
{
  typedef CTT_Parser< -->CharType, -->PluginType >  inherited;
public:

  typedef CTT_Node<char_type>          node;
  typedef CTT_Map<str, bool >          mstrbool;
  typedef CTT_Map<str, int >           mstrint;
  typedef CTT_Map<str, unsigned int >  mstruint;
  typedef CTT_Map<str, char >          mstrchar;
  typedef CTT_Map<str, str >           mstrstr;
  typedef CTT_Map<str, node >          mstrnode;
  typedef CTT_Vector< bool >           vbool;
  typedef CTT_Vector< int >            vint;
  typedef CTT_Vector< unsigned int >   vuint;
  typedef CTT_Vector< char >           vchar;
  typedef CTT_Vector< str >            vstr;
  typedef CTT_Vector< node >           vnode;


  -->ScannerEnum

  -->ParserClassName();

-->StartRuleDeclaration
-->InterfaceDeclarations

private:

-->ParserRuleDeclarations
-->InitProcDeclaration

};

} // namespace tetra


#endif // -->ParserHeaderSentinel
```

Mit dem Pfeil "-->" und anschließendem Schlüsselwort sind die Positionen gekennzeichnet, an die TETRA seinen Code schreibt.

-->ParserHeaderSentinel wird bei der Codeerzeugung durch einen Ausdruck ersetzt, der aus dem Namen der Startregel des Parser gebildet wird. Ist beispielsweise der Name der Startregel: "Pascal", so wird der gesamte Sentinel folgendermaßen aussehen:

```
#ifndef PascalparserH
#define PascalparserH
…
#endif // PascalparserH
```

-->ScannerEnum bezeichnet den Ort, an den Enumerationstypen geschrieben werden. Die einzelnen Enumerationswerte sind Indizes für die verschiedenen Mini-Scanner.

-->ParserClassName steht für den Namen der Parserklasse

Das zweite Vorkommen von **-->ParserClassName** bezeichnet den Konstruktor der Parserklasse. In die anschließende Klammer können evtl. zusätzliche Parameter eingefügt werden.

StartRuleDeclaration bezeichnet die öffentliche Klassenfunktion zum Aufruf des Parsers, der aus dem Namen der Startregel gebildet ist, z.B.:

```
void Pascal(cts xtBegin, cts xtEnd);
```

ctsr ist ein typedef für std::string::const_iterator.

InitProcDeclaration steht für die Deklaration:

```
void Init();
```

Damit wird die Prozedur zur Initialisierung der Klassenvariablen bezeichnet:

InterfaceDeclarations steht für die Reihe der Deklarationen von Funktionen zur direkten Ausführung von anderen als der Startregel. In den lokalen Optionen muss *Interfacemethode erzeugen* aktiviert werden, um die Erzeugung dieser Interface-methoden zu veranlassen. .Die Deklarationen haben die gleiche Form, wie die Deklaration der Startregel.

ParserRuleDeclarations schließlich bezeichnet die Position an der die Liste der Deklarationen für die einzelnen Produktionen und für Unterklassen zum Scannen des Inputs eingefügt werden. Jeder Production entspricht so einer gleichnamigen Klassenfunktion der Parserklasse. Insbesondere die Startregel selbst wird hier deklariert:

```
"void Pascal(tetra::sps& xState);"
```

sps ist ein typedef für CTT_ParserState, eine Klasse die den Zustand, d.h. insbesondere die Positionen des Parseprozesses enthält, und die als Parameter von Regel zu Regel weitergereicht wird, wobei jeweils der Zustand aktualisiert wird.

13.1.3 Implementations-Schablone

Die unmodifizierte Schablone für die Implementation einer TETRA-Parserklasse ist folgende:

```
//-----------------------------------------------------------
//   ttparser_cpp.frm
//   TextTransformer C++ Support Frame
//   Copyright: Dr. Detlef Meyer-Eltz
//   http://www.texttransformer.de
//   http://www.texttransformer.com
//   Meyer-Eltz@t-online.de
//
//   March, 2006  Version 1.1.0
//-----------------------------------------------------------


#ifdef __BORLANDC__
#pragma hdrstop
#endif


#include "-->ParserHeaderName"
#include <iostream>
#include "tt_exception.h"
#include "tt_guard.h"
#include "tt_localscanner.h"
#include "tt_parsestateplugin.h"
#include "tt_scanner.h"

#define indent xState.GetPlugin()->GetIndentPtr()

using namespace std;
using namespace tetra;

// return types
typedef -->ParserClassName::node node;
typedef -->ParserClassName::str str;


-->TokenList


-->ParserClassName::-->ParserClassName()
-->MemberInitialization
{
  try
  {
    CreateScannerArray(eScannerLast);
    Init();
  }
  catch(boost::regex_error& xErr)
```

```
  {
    cleanup();
    throw CTT_Message(xErr.what());
  }
  catch(CTT_Message& eMsg)
  {
    cleanup();
    // did you use the actual token file?
    throw eMsg;
  }
  catch(...)    // std::bad_alloc
  {
    cleanup();
    // did you use the actual token file?
   throw CTT_Message("parser creation failed");
  }
}

-->StartRuleHeading
{
-->StartRule
}

-->InitProcImplementation

-->InterfaceImplementations

-->ParserRules
```

-->ParserHeaderName wird zu *Pascalparser.h*, um bei dem Beispiel aus dem vorherigen Abschnitt zu bleiben

-->**TokenList** wird durch eine auskommentiert Liste der Namen und Definitionen der Token ersetzt. Diese Liste dient lediglich dem besseren Verständnis des nachfolgenden Codes.

-->**ParserClassName** hat dieselbe Bedeutung wie im vorherigen Abschnitt beschrieben. In den Klammerausdruck hinter dem zweiten Vorkommen von ParserClassName können die der Deklaration entsprechenden Parameter geschrieben werden, falls auch die Header-Schablone entsprechend modifiziert ist.

-->**MemberInitialization**, hier werden Klassenvariablen - z.B. Funktionstabellen -initialisiert.

-->**StartRuleHeading** entspricht der **ParserCallDeclaration** der Header-Schablone. Dies ist der Kopf der Aufruffunktion des Parsers.

-->**StartRule** reserviert den Platz für den Funktionsblock der Aufruffunktion des Parsers.Er kann bei Bedarf von einem try-catch-Block eingeschlossen werden.

-->**InitProcImplementation** ist der Ort an dem die Prozedur zur Initialisierung der Klassenvariablen

eingefügt wird.

-->**InterfaceImplementations** steht für die Reihe der Funktionen zur direkten Ausführung von anderen als der Startregel. In den lokalen Optionen muss "Interfacemethode erzeugen" aktiviert werden, um die Erzeugung dieser Interface-methoden zu veranlassen. .

-->**ParserRules** schließlich ist die Position an die die Implementierungen der Parserregeln geschrieben werden.

13.1.4 main-Datei-Schablone

Standardmäßig wird mit der folgenden main-Datei-Schablone eine main-Funktion für ein Konsoleprogramm erzeugt.

```
//-------------------------------------------------------------
//    ttmain_c.frm
//    TextTransformer C++ Support Frame
//    Copyright: Dr. Detlef Meyer-Eltz
//     http://www.texttransformer.de
//     http://www.texttransformer.com
//     dme@texttransformer.com
//
//    June, 2009   Version 1.7.0
//-------------------------------------------------------------


#ifdef  __BORLANDC__
#pragma  hdrstop
#endif


#include   "-->ParserHeaderName"
-->XercesInclude



using  namespace std;
using  namespace tetra;
-->XercesUsingNamespace


typedef -->ParserClassName::string_type   string_type;

void usage()
{
     cout << "\nParameters:\n"
             "  -s    source file\n"
             "  -t    target file\n"
          << endl;
}

int main(int argc, char* argv[])
{
   string_type sTest;
  const char* pSourceName = NULL;
  const char* pTargetName = NULL;
```

```
if (argc < 2)
{
   usage();
  return 1;
}

int iParam;
for(iParam = 1; iParam < argc; iParam++)
{
  if(argv[iParam][0] != '-')
  {
     usage();
    return 2;
  }

  if(!strcmp(argv[iParam],   "-s"))
  {
     if(++iParam < argc)
        pSourceName = argv[iParam];
  }

  if(!strcmp(argv[iParam],   "-t"))
  {
    if(++iParam < argc)
        pTargetName = argv[iParam];
  }
}

if(pSourceName == NULL   )
{
   usage();
  return 3;
}

if(pTargetName == NULL   )
{
   usage();
  return 4;
}

if( !-->LoadFile )
{
    cout << "could not load source file: " << pSourceName;
    return 5;
}

-->Ostream
if(!fout)
{
   cout << "could not open target file: " << pTargetName;
  return 6;
}

-->PluginType  plugin(fout);
 InitPluginPaths(plugin,  pSourceName,  pTargetName);
 plugin.UseExcept(true);

-->XercesInit
-->ParserClassName Parser;
```

```
try
{
   Parser.-->StartRuleName(sTest.begin(),  sTest.end(),  &plugin);
}
 catch(CTT_ErrorExpected&  xErr)
{
     cout << "expected: "
          << xErr.GetWhat()
         << " in "
          << xErr.GetProduction()
         << "\n";
}
 catch(CTT_ErrorUnexpected&  xErr)
{
     cout << "unexpected token in: "
          << xErr.GetProduction()
         << "\n";
}
 catch(CTT_ParseError&  xErr)
{
     cout << xErr.GetWhat();
}
 catch(CTT_Message&  xErr)
{
   cout << xErr.what();
}
/*
 catch(boost::system_error&  xErr)
{
    cout << xErr.what();
}*/
 catch(exception&  xErr)
{
    cout << xErr.what();
}
/* xerces catches
 catch  (const  OutOfMemoryException&)
{
   cout << ccpXercesOutOfMemory;
}
 catch  (const  DOMException& e)
{
   cout << "xerces error code: " << e.code << endl;
    char  *pMsg  =  XMLString::transcode(e.getMessage());
   cout << pMsg;
    XMLString::release(&pMsg);
}
 catch  (XMLException& e)
{
    char  *pMsg  =  XMLString::transcode(e.getMessage());
   cout << pMsg;
    XMLString::release(&pMsg);
}
*/

 if(  plugin.HasMessage())
{
  -->PluginType::ctvmsg  t,  tEnd  =  plugin.MsgEnd();
   for(t  =  plugin.MsgBegin();  t  !=  tEnd;  ++t)
```

```
        cout << (*t).what() << endl << endl;
    }

    return 0;
}
```

Das Programm erwartet zwei Parameter: einen für die Quelltextpfad und einen für den Zielpfad.

In der Schablone gibt es fünf neue Platzhalter, die in den anderen Schablonen nicht vorkommen.

XercesInclude
XercesUsingNamespace
XercesInit
LoadFile
Ostream

Die ersten drei Platzhalter werden gelöscht, wenn im Projekt keine dnode's verwendet werden.

-->**LoadFile** wird je nach Modus zum Öffnen der Quelldatei ersetzt durch:

```
load_file_binary( sTest, pSourceName)
```

oder

```
load_file( sTest, pSourceName)
```

-->**Ostream** wird entweder durch *wofstream* oder durch *ofstream* ersetzt, je nachdem, ob die Wide-Zeichen Option aktiviert ist oder nicht und je nach Modus zum Öffnen der Zieldatei wird das Flag *ios::binary* gesetzt oder nicht.

Wenn dnode's verwendet werden, muss als Plugin-Typ CTT_ParseStateDomPlugin gesetzt werden.

-->**XercesInclude** sorgt für den Einschluss von tt_xerces.h

->**XercesUsingNamespace** fügt folgende Zeile ein:

```
using namespace xercesc;
```

-->**XercesInit** erzeugt eine *CTT_Xerces* Instanz und übergibt einen Zeiger auf sie an das Plugin:

```
CTT_Xerces Xerces("root", "UTF-8", false, true, true, true);
//Xerces.setDTDParams("", "", "");
 plugin.SetXerces(&Xerces);
   Ctranslation_unitParser::dnode::SetDefaultLabel(L"default_label");
```

13.1.5 Projektspezifische Schablonen

Für jedes Projekt können nach Bedarf eine eigene spezifische Schablonen erstellt werden. Ihr Speicherort wird in den Projekteinstellungen angegeben. Hier können beispielsweise Klassenvariablen oder -methoden definiert werden und zusätzliche Include-Direktiven eingefügt werden.
Hierbei können die oben erläuterten Platzhalter benutzt werden. Z.B. könnte ein globaler Zeiger auf eine Instanz des Parsers folgendermaßen definiert werden:

extern -->ParserClassName* _-->ParserClassName;

Wichtig sind hierbei, dass ein nicht alphanumerisches Zeichen hinter "**-->ParserClassName**" steht ohne die dieser Platzhalter ignoriert würde.

13.1.6 jamfile

Bei der Generierung des C++-Codes wird zusätzlich wird eine Datei mit dem Namen *jamfile.txt* generiert, die als Grundlage für ein "jamfile" zur Verwendung mit dem Boost Build System dienen kann.

http://www.boost.org/doc/tools/build/index.html

Nachfolgend ist ein Beispiel für solch eine Datei wiedergegeben, die für ein Projekt mit der Startregel *translation_unit* erzeugt wurde, dass dnode's verwendet.

```
# frame for a bjam jamfile

lib regex : : <name>[name of the regex lib] <search>[search path for the regex lib] ;
lib filesystem : : <name>[name of the filesystem lib] <search>[search path for the filesystem
lib xerces : : <name>[name of the xerces lib] <search>[search path for the xerces lib] ;

CPP_SOURCES =
tt_boost_config tt_exception tt_lib tt_msg tt_node tt_domnode tt_xerces
;

exe translation_unit : [path of the TextTransformer code]/$(CPP_SOURCES).cpp [ glob *.cpp ] fi
          : <include>[path to the boost directoy] // e.g. C:/Program Files (x86)/boost/boost_1_38
            <include>[path to the TextTransformer code] // e.g. C:/Program Files (x86)/TextTrans
            <include>[path to the xerces source] // e.g. C:/Program Files (x86)/xerces-c-3.0.1/sr
;
```

Wenn CTT_ParseStateDomPlugin nicht als als Plugin-Typ gesetzt ist, fehlen die xerces betreffenden Zeilen in *jamfile.txt*. Auch die boost filesystem-Bibliothek ist nicht immer erforderlich. Das wird aber nicht automatisch geprüft.

13.2 Unterstützender Code

Zur Kompilierung eines von TETRA erzeugten Parsers ist der Quellcode für einige zusätzliche Klassen erforderlich, die in den Projekten verwendet werden. Die meisten Klassen sind allein in Header-Dateien untergebracht ("header-only"), d.h. ihr Quellcode muss nicht explizit in die Projekte des C++-Compilers eingebunden werden. Die Struktur des tetra-Verzeichnisses zeigt deutlich, welche Dateien eingebunden werden müssen, nämlich diejenigen, die sich im *tetra/source* -Verzeichnis befinden.

Die vollständige Liste der Klassen ist:

CTT_Error
CTT_ParseError
CTT_Parser
CTT_ParseState
CTT_Scanner
CTT_Tst
CTT_Match
CTT_Guard
CTT_Mstrstr
CTT_Mstrfun
CTT_Node
CTT_DomNode
CTT_ParseStatePluginAbs
CTT_ParseStatePlugin
CTT_ParseStateDomPluginAbs
CTT_ParseStateDomPlugin
CTT_RedirectOutput
CTT_Indent
CTT_Xerces

Eine Implementation der Hilfsfunktionen

stod
stoi
dtos
itos
etc.

befindet sich in der Datei tt_lib.h/cpp

13.2.1 Verzeichnisstruktur

Der unterstützende Code für TextTransformer Parser befindet sich im Verzeichnis *tetra*. Die Struktur des Verzeichnis orientiert sich am Modell der *boost*-Bibliotheken und spiegelt wieder, welche Dateien in die Projekte des C++-Compiler eingebunden werden müssen.

```
tetra
```

```
|
 ---  config
|
 ---  source
|       |
|          ---  xercesdom
|
 ---  xercesdom
```

Eingebunden werden müssen nur die Dateien des Unterverzeichnisses *source* und bei Verwendung von dnode's auch das Verzeichnis *source/xercesdom*. Alle Dateien im Wurzelverzeichnis *tetra* und im Verzeichnis *tetra/xercesdom* sind Header-Dateien.

13.2.2 CTT_Parser

template <class char_type> **class** CTT_Parser

CTT_Parser ist eine einfache Basisklasse für die von TETRA generierten Parser. Der Template-Parameter kann entweder *char* oder *wchar_t* sein.
Bei Bedarf kann vom generierten Parser wiederum eine eigen Klasse abgeleitet werden, in der die virtuellen Methoden von *CTT_Parser* überschrieben werden können, z.B. die Parser-Ereignisse.

13.2.2.1 Methoden

Die virtuellen Methoden der Klasse *CTT_Parser* sind in generierten Parsern teilweise überschrieben oder können vom Entwickler in einer eigenen Klasse überschrieben werden.

```
virtual  int              GetNext( state_type& xState,
                                  int xiNode,
                                   int xiScannerIndex = -1,
                                   int xiSkipScannerIndex = -1,
                                  ELState xe = eDefault,
                                   const char* xpProduction = NULL,
                                   const char* xpSymbol = NULL);

virtual  int              ConstGetNext( state_type& xState,
                                       int xiNode,
                                        int xiScannerIndex = -1,
                                        int xiSkipScannerIndex = -1,
                                       ELState xe = eDefault,
                                        const char* xpProduction = NULL,
                                        const char* xpSymbol = NULL) const;

virtual  bool             LA_GetNext( state_type& xState,
                                     int xiNode,
                                      int xiScannerIndex = -1,
                                      int xiSkipScannerIndex = -1,
                                     ELState xe = eDefault,
                                      const char* xpProduction = NULL,
                                      const char* xpSymbol = NULL) const;
```

```
virtual  int              GetStartSym( state_type& xState,
                                       int xiScannerIndex,
                                       int xiSkipScannerIndex,
                                      int xiNode,
                                       const char* xpProduction = NULL,
                                       const char* xpSymbol = NULL);

virtual  int              ConstGetStartSym( state_type& xState,
                                            int xiScannerIndex,
                                            int xiSkipScannerIndex,
                                           int xiNode,
                                            const char* xpProduction = NULL,
                                            const char* xpSymbol = NULL) const;

virtual  int              LA_GetStartSym( state_type& xState,
                                           int xiScannerIndex,
                                           int xiSkipScannerIndex,
                                          int xiNode,
                                           const char* xpProduction = NULL,
                                           const char* xpSymbol = NULL) const;



virtual  bool             AcceptAndGetNext( state_type& xState,
                                           int xiSym,
                                            int xiScannerIndex,
                                            int xiSkipScannerIndex,
                                           ELState xe,
                                            const char* xpProduction = NULL,
                                            const char* xpSymbol = NULL);

virtual  bool             LA_AcceptAndGetNext( state_type& xState,
                                              int xiSym,
                                               int xiScannerIndex,
                                               int xiSkipScannerIndex,
                                              ELState xe,
                                               const char* xpProduction = NULL,
                                               const char* xpSymbol = NULL) const;

virtual  bool             ConstAcceptAndGetNext( state_type& xState,
                                                 int xiSym,
                                                  int xiScannerIndex,
                                                  int xiSkipScannerIndex,
                                                 ELState xe,
                                                  const char* xpProduction = NULL,
                                                  const char* xpSymbol = NULL) const;
virtual  bool             Accept(  state_type& xState,
                                  int xiSym,
                                   const char* xpProduction = NULL,
                                   const char* xpSymbol = NULL);

virtual  bool             ConstAccept( state_type& xState,
                                       int xiSym,
                                        const char* xpProduction = NULL,
                                        const char* xpSymbol = NULL) const;

virtual  bool             AcceptSkip( state_type& xState,
                                     int xiSym,
                                      const char* xpProduction = NULL,
```

```
                                    const char* xpSymbol = NULL);

virtual  bool             ConstAcceptSkip( state_type& xState,
                                           int xiSym,
                                            const char* xpProduction = NULL,
                                            const char* xpSymbol = NULL) const;


virtual  bool             LA_Accept( state_type& xState,
                                     int xiSym,
                                      const char* xpProduction = NULL,
                                      const char* xpSymbol = NULL) const;

virtual  int               CheckInclusion(state_type& xState, int xiNode, const char* xpProducti
                          {return 0;}

virtual  int               ConstCheckInclusion(state_type& xState, int xiNode, const char* xpPro
                          {return 0;}

virtual  int               LA_CheckInclusion(state_type& xState, int xiNode, const char* xpProdu
                          {return 0;}

virtual GUARD             ProductionBegin(state_type& xState,
                                          int xiScannerIndex,
                                          int xiSkipScannerIndex,
                                         ELState xe,
                                          int xiProdIndex,
                                          const char* xpProduction);

virtual GUARD             ConstProductionBegin(state_type& xState,
                                               int xiScannerIndex,
                                               int xiSkipScannerIndex,
                                              ELState xe,
                                               int xiProdIndex,
                                                const char* xpProduction) const;

virtual GUARD             LA_ProductionBegin(state_type& xState,
                                             int xiScannerIndex,
                                             int xiSkipScannerIndex,
                                            ELState xe,
                                             int xiProdIndex,
                                              const char* xpProduction) const;

virtual  void             OnErrorExpected(state_type& xState,
                                          int xiSym,
                                          const char* xpProduction,
                                          const char* xpBranch);

virtual  void             OnErrorExpected(state_type& xState,
                                          int xiSym,
                                          const char* xpProduction,
                                          const char* xpBranch) const;

virtual  void             OnErrorSkipExpected(state_type& xState,
                                              int xiSym,
                                               const char* xpProduction,
                                               const char* xpBranch);

virtual  void             OnErrorSkipExpected(state_type& xState,
                                              int xiSym,
```

```
                                          const char* xpProduction,
                                          const char* xpBranch) const;

virtual  void            OnErrorUnexpected(state_type& xState,
                                           const char* xpProduction,
                                           const char* xpBranch);

virtual  void            OnErrorUnexpected(state_type& xState,
                                           const char* xpProduction,
                                           const char* xpBranch) const;

virtual  void            OnErrorStandstill(state_type& xState,
                                           const char* xpProduction,
                                           const char* xpBranch);

virtual  void            OnErrorStandstill(state_type& xState,
                                           const char* xpProduction,
                                           const char* xpBranch)  const;

virtual  void            OnErrorIncomplete(state_type& xState,
                                           const char* xpProduction,
                                           const char* xpBranch);

virtual  void            OnErrorIncomplete(state_type& xState,
                                           const char* xpProduction,
                                           const char* xpBranch) const;

virtual  void           AddMessage(state_type& xState,
                                   const string_type& xs) const;

virtual  void           AddWarning(state_type& xState,
                                   const string_type& xs) const;

virtual  void           AddError(state_type& xState,
                                 const string_type& xs) const;

virtual  void           GenError(state_type& xState,
                                 const string_type& xs) const;

virtual  void           AddMessage(state_type& xState,
                                   const string_type& xs,
                                    difference_type xuiLastPosition,
                                    difference_type xuiPosition,
                                   const char* xpProductionm,
                                   const char* xpSymbol,
                                   EMsgType xeMsgType) const;


virtual  void            SourceName(const string_type& xsSourcename,
                                    bool xbIsLastFile = true);
virtual  void           TargetName(const string_type& xsTargetname);
virtual  void           SourceRoot(const string_type& xsSourceRoot);
virtual  void           TargetRoot(const string_type& xsTargetRoot);

virtual  void           AddDynamicTokens(plugin_ptr_type xpPlugin) const {}  // has to be ove

virtual  void           OnAcceptToken(state_type& xState) {/* to be overwritten */};
virtual  void           OnAcceptToken(state_type& xState) const {/* to be overwritten */};
virtual  void           OnParseError(state_type& xState) {/* to be overwritten */};
virtual  void           OnParseError(state_type& xState) const {/* to be overwritten */};
```

```
virtual  void             OnEnterProduction(state_type& xState) {/* to be overwritten */};
virtual  void             OnEnterProduction(state_type& xState) const {/* to be overwritten */}
virtual  void             OnExitProduction(state_type& xState) {/* to be overwritten */};
virtual  void             OnExitProduction(state_type& xState) const {/* to be overwritten */};

virtual  bool            IsDone(state_type& xState, int xi) const;
```

13.2.3 CTT_ParseState

template <class char_type, class plugin_type> class CTT_ParseState

CTT_ParseState repräsentiert den aktuellen Zustand eines Parsers. Insbesondere enthält die Klasse ein boost::match_results<iterator>-Element, das die Informationen über den letzten Treffer enthält, sowie diverse Iteratoren, die die Positionen im Quelltext markieren.
Der 1. Template-Parameter kann entweder *char* oder *wchar_t* sein. Der 2. Template-Parameter ist entweder CTT_ParseStatePlugin oder ein davon abgeleiteter Typ.

13.2.4 CTT_Scanner

template <class char_type> **class** CTT_Scanner

Instanzen der Klasse CTT_Scanner organisieren die Ermittlung des jeweils nächsten Tokens and den Entscheidungsstellen des Parsers. Der Template-Parameter kann entweder *char* oder *wchar_t* sein.

CTT_Scanner verteilt seine Aufgabe über lokale Scanner:
CTT_IgnoreScanner: entfernt überflüssige Zeichen
CTT_LiteralScanner: testet auf Literale
CTT_DynamicScanner: testet auf dynamische Literale
CTT_RegexScanner: testet auf reguläre Ausdrücke
CTT_SkipScanner: testet auf das nächste Vorkommen eines Tokens im Text

13.2.5 CTT_Tst, CTT_TstNode

template <class char_type> **class** CTT_Tst

CTT_Tst ist die Implementation eines ternären Suchbaums für literale Token.
CTT_TstNode's bilden die Knoten des Suchbaums.
Der Template-Parameter kann entweder *char* oder *wchar_t* sein.

13.2.6 CTT_Match

template <**class** char_type > **class** CTT_Match
 : **public** CTT_Token<char_type>

CTT_Match ist eine Klasse, die das Ergebnis des Tests eines Tokens am Text festhält. Im Unterschied zur Basisklasse CTT_Token enthält CTT_Match ein *match_results* Element, das in der boost-Bibliothek für reguläre Ausdrücke definiert ist.

Der Template-Parameter kann entweder *char* oder *wchar_t* sein.

13.2.7 CTT_Token

template <**class** char_type > **class** CTT_Token

CTT_Token ist eine Klasse, die das Ergebnis des Tests eines Tokens am Text festhält. Im Unterschied zur abgeleiteten Klasse CTT_Match enthält sie kein *match_results* Element, und wird somit zum effizienten speichern der Token im Token-Puffer verwendet.

13.2.8 CTT_Buffer

template <**typename** char_type> **class** CTT_BufferAbs

template <**typename** char_type> **class** CTT_BufferBase : **public** CTT_BufferAbs<char_type>

template <**typename** char_type> **class** CTT_BufferLL1ex : **public** CTT_BufferBase<char_type>

template <**typename** char_type > **class** CTT_BufferAll : **public** CTT_BufferBase<char_type>

Beim Parsen eines Textes sind mindestens das zuletzt erkannte Token und das nächste erkannte Token gepuffert. Die Pufferung geschieht mit diesen Klassen.

CTT_BufferLL1ex enthält drei CTT_Match-Elemente für die beiden genannten Token und ein eventuell erkanntes Token, zu dem mit SKIP gesprungen wird.

CTT_BufferAll enthält einen Stack aus CTT_Token zur Pufferung der Vorausschau-Token.

Der Template-Parameter kann entweder *char* oder *wchar_t* sein.

13.2.9 CTT_Guard

template <class char_type, class plugin_type> **class** CTT_Guard

Zu Beginn einer jeden Produktion sorgt der Konstruktor einer Instanz der Klasse CTT_Guard dafür, dass ein Stack aktualisiert wird, der aus den Scannern besteht - genauer: aus Zeigern auf die Scanner -, die am Ende des Aufrufs der Produktion zu testen sind. Beim Verlassen der Produktion sorgt der Destruktor von CTT_Guard für die erneute Aktualisierung des Stacks, indem der zu Beginn hinzugefügte Scanner wieder entfernt wird. Der Destruktor wird benutzt, da andere Anweisungen nach einer return-Anweisung nicht mehr ausgeführt würden.

Der 1. Template-Parameter kann entweder *char* oder *wchar_t* sein. Der 2. Template-Parameter ist entweder CTT_ParseStatePlugin oder ein davon abgeleiteter Typ.

Makros:

Der vom TextTransformer produzierte Code soll portabel sein, d.h.er soll auf verschiedenen Systemen und mit verschiedenen C++-Kompilern funktionieren.
Wegen einer Besonderheit von *Microsoft Visual Express C++* ist es erforderlich den gesamten Code einer Produktion mittels zweier Makros in einen *try-catch*-Block einzuschließen, damit der Aufruf des Destruktors tatsächlich zum gewünschten Zeitpunkt erfolgt.

```
#define  ENTER_GUARD(number,  production)  \
   GUARD  Guard  =  ProductionBegin(xState,  xiScannerIndex,  xiSkipScannerIndex,  xeLS,  number,  produ
  try {
```

(ENTER_CONST_GUARD and ENTER_LA_GUARD sind entprechende Makros für const-Parser und Vorausschauen.)

```
#define  EXIT_GUARD(number,  returnvalue)  \
       } \
  catch (...) { \
      if(xiScannerIndex > -2) \
         throw; \
  } \
   Guard.StayAlive(); \
   return  returnvalue;
```

In *Visual Express C++* sorgt nämlich eine "Optimierung" dafür, dass die *CTT_Guard*-Variable sofort nach ihrer Erzeugung wieder aufgeräumt wird. Um die Variable bis zum Verlassen der Produktion am Leben zu erhalten, muss dem *Microsoft*-Compiler vorgegaukelt werden, dass sie nochmals benötigt würde. Deshalb wird am Ende der Produktion, nach dem *catch*-Block, nochmals die ansonsten funktionslose *StayAlive*-Funktion der *CTT_Guard*-Klasse aufgerufen. Diese Funktion wird nur durchlaufen, wenn die Produktion keinen Wert zurück gibt. Andernfalls erfolgt die Rückgabe schon innerhalb des try-catch-Blocks. Wenn tatsächlich eine Ausnahme ausgeworfen wird, wird sie mit *throw* auf jeden Fall weitergereicht. Die Bedingung:

```
   if(xiScannerIndex  >  -2)
```

ist durch den Code-Generator immer erfüllt.Dem Compiler gilt der *StayAlive*-Aufruf aber als erreichbar.

Die letzte Zeile des Makros ist erforderlich, damit der Code kompiliert:

```
return  returnvalue;
```

Wenn die Produktion keinen Wert zurück gibt bleibt *returnvalue* leer.Wenn die Produktion aber einen Wert zurück gibt, muss ein Default-Wert für den Rückgabetyp bekannt sein, obwohl die Zeile tatsächlich niemals ausgeführt wird. Dieser Wert kann daher beliebig sein, solange er nur zum entsprechenden Rückgabetyp passt. Für den Interpreter-Code wird der Rückgabewert automatisch generiert. Wenn aber in Code, der nur für den Export bestimmt ist ein Rückgabetyp definiert ist, muss der Default-Wert angegeben werden. Dies kann im Feld für den Rückgabetyp geschehen, indem ein Schrägstrich mit nachfolgendem Wert geschrieben wird. Z.B.:

```
{_  CProduktion*  _}/NULL
```

13.2.10 CTT_Mstrstr

template <class char_type> **class** CTT_Mstrstr

CTT_Mstrstr verfügt über die gleichen Schnittstellen wie mstrstr des Interpreters. Im erzeugten Code ist mstrstr definiert durch:

typedef CTT_Mstrstr<char> mstrstr;

CTT_Mstrstr ist abgeleitet von std::map<Key, T, Compare, Allocator > verfügt so auch über sämtliche Schnittstellen einer std::map.

13.2.11 CTT_Mstrfun

template <class char_type, class object_pointer, class return_type, class memfun_pointer> **class** CTT_Mstrfun

CTT_Mstrstr ist abgeleitet von std::map<Key, T, Compare, Allocator >. Mit dieser Klasse können Zuordnungen von Namen und Zeigern auf Parser-Methoden gepeichert werden.

13.2.12 CTT_Node

template <class char_type> class CTT_Node

CTT_Node ist eine eigens für den TextTransformer geschriebene Klasse, die den Type node im exportierten C++-Code repräsentiert, und die im Code des TextTransformers selbst verwendet wird. Der Template-Parameter kann entweder *char* oder *wchar_t* sein.

13.2.13 CTT_DomNode

template <class char_type> class CTT_DomNode;

Die dnode Knoten des Interpreters sind im erzeugten Code als typedef von CTT_DomNode definiert.
CTT_DomNode kapselt ein xercesc DOMElement mit einem xercesc DOMText Kind.
CTT_DomNode kann so auf die gleiche Weise verwendet werden und hat die gleichen Funktionen wie CTT_Node, Der Name des DOMElements repräsentiert das Label des Knotens und der Wert des DOMTexts den Wert des Knotens. Wird ein CTT_DomNode-Knoten in eine XML-Datei geschrieben, erhält man:

<label>value</label>

Während der Speicher einer CTT_Node intern durch Referenz-Zählung aller Knoten eines Baums verwaltet wird, wird eine CTT_DomNode durch xerces verwaltet.

Vor dem Aufruf des Parsers im generierten C++-Code, muss das Default-Label manuell gesetzt werden. Die *CTT_DomNode* Klasse hat hierfür eine statische Methode;

```
dnode::SetDefaultLabel(L"default_label");
```

13.2.14 CTT_ParseStatePluginAbs

template <class char_type >
CTT_ParseStatePluginAbs<char_type>;

Abstrakte Basisklasse für CTT_ParseStatePlugin und für CTT_ParseStateDomPluginAbs.

13.2.15 CTT_ParseStatePlugin

template <class char_type> class CTT_ParseStatePlugin

CTT_ParseStatePlugin ist die Klasse für das Plugin. Von dieser Klasse können benutzerspezifische Klassen abgeleitet werden, die Methoden und Daten enthälten, die während genau eines Parserdurchlaufs gültig sind.
Der Template-Parameter kann entweder *char* oder *wchar_t* sein.
Der Plugin-Typ muss in den Projekteinstellungen gesetzt werden.

13.2.16 CTT_ParseStateDomPluginAbs

template <class char_type >
class CTT_ParseStateDomPluginAbs : public CTT_ParseStatePluginAbs<char_type>;

13.2.17 CTT_ParseStateDomPlugin

template <class char_type> class CTT_ParseStateDomPlugin

Diese Klasse enthält die gleichen Daten und Funktionen wie CTT_ParseStatePlugin, zusätzlich aber noch einen Zeiger auf ein CTT_Xerces, das ein xerces DOMDocument verwaltet.
Der Plugin-Typ muss in den Projekteinstellungen gesetzt werden. Wenn dnode Knoten im Projekt verwendet werden, muss CTT_ParseStateDomPlugin verwendet werden und die Xerces Bibliothek muss zum erzeugten Code gelinkt werden..

13.2.18 CTT_RedirectOutput

template <class char_type > **class** CTT_RedirectOutput

Diese Klasse verwaltet die Umleitung der Ausgabe in Dateien.
Der Template-Parameter kann entweder *char* oder *wchar_t* sein.

13.2.19 CTT_Indent

template <class char_type > class CTT_Indent

Ein einfacher Einrückungs-Stack. Der Template-Parameter kann entweder *char* oder *wchar_t* sein.

13.2.20 CTT_Xerces

class CTT_Xerces

Diese Klasse kapselt ein xerces DOMDocument und enthält Funktionen zu dessen Verwendung. Ein Zeiger auf eine Instanz dieser Klasse ist in CTT_ParseStateDomPlugin enthalten. Das Dokument kann daher im Parser erzeugt werden und dann außerhalb desselben benutzt werden.

```
  CTT_Xerces();
  CTT_Xerces(const  std::string&  xsRootElementName,
             const  std::string&  xsEncoding,
             bool xbWriteBOM = false,
             bool xbPrettyPrint = true,
             bool xbWriteDOMDeclaration = true,
             bool xbStandalone = true);
 ~CTT_Xerces();

  XERCES_CPP_NAMESPACE::DOMDocument*  GetDocument();

void           setDTDParams(const  std::string&  xsName,
                            const  std::string&  xsPublicID,
                            const  std::string&  xsSystemID);

bool     createDocument();
void     destroyDocument();
bool      writeToFile(const  std::string&  xsFilename);
```

```
bool      writeToString(std::string&  xsResult);
bool      writeToString(std::wstring&  xsResult);
bool      writeToStream(std::ostream&  xos);
bool      writeToStream(std::wostream&  xos);
bool     hasDOM() const;
```

13.3 Fehlerbehandlung

Funktionen zur Fehlerbehandlung sind bei den Plugin-Methoden zu finden. Darüber hinaus gibt es vier Ereignisse, die den Parser zum Abbruch zwingen und für jedes dieser Ereignisse gibt es eine Behandlungsroutine, die im Normalfall einen entsprechenden Fehlertyp auswirft. Diese Methoden sind virtuell und können überschrieben werden.

```
void    OnErrorExpected(state_type&  xParseState,
                        int xiSym,
                        const char* xpProduction,
                        const char* xpBranch) const;
```

Das Token mit der Nummer *xiSym* wurde in der Produktion mit dem Namen *xpProduction* an der Verzweigung mit dem Namen *xpBranch* erwartet, jedoch wurde es nicht gefunden.

```
void    OnErrorUnexpected(state_type&  xParseState,
                          const char* xpProduction,
                          const char* xpBranch) const;
```

Ein anderes Token wurde in der Produktion mit dem Namen *xpProduction* an der Verzweigung mit dem Namen *xpBranch* erwartet.

```
void    OnErrorIncomplete(state_type&  xParseState,
                          const char* xpProduction,
                          const char* xpBranch) const;
```

Der Eingabetext konnte mit der Produktion *xpProduction* nicht vollständig geparst werden.

```
void    OnErrorStandstill(state_type&  xParseState,
                          const char* xpProduction,
                          const char* xpBranch) const;
```

Der Parser befindet sich in der Produktion mit dem Namen *xpProduction* an der Verzweigung mit dem Namen *xpBranch* in einer Endlosschleife. (Dieser Fehler kann zur zeit nicht auftreten, da nur Token zugelassen sind, die mindestens ein Zeichen abdecken.)

CTT_Message ist Basisklasse für alle Meldungen, Warnungen Fehler und Ausnahmen. Neben der eigentlichen Meldung enthält sie Angaben zum aktuellen Zustand des Parsers.

```
CTT_Message(const  std::string&  xsWhat,
            unsigned int xuiLastPosition,
            unsigned int xuiPosition,
```

```
const char* xpProduction,
const char* xpSymbol = NULL,
EMsgType xeMsgType = eMessage,
unsigned int xui = 0)
```

Von CTT_Message direkt oder indirekt abgeleitet sind:

CTT_Exit
CTT_Warning
CTT_Error
CTT_ParseError
CTT_ErrorExpected
CTT_ErrorUnexpected
CTT_ErrorStandstill
CTT_ErrorIncomplete
CTT_SemError
CTT_NodeError

13.4 Compiler-Kompatibilität

Der generierte C++-Code ist im wesentlichen portabel. Er wurde mit folgenden Compilern erfolgreich getestet:

Windows

Borland CBuilder 6
Borland CodeGear C++ Builder 2009
Visual C++ 2008 Express

Linux

gcc 4.2

Mit anderen Compilern und unter anderen Systemen könnten eventuell kleinere Anpassungen der Code-Schablonen und des unterstützenden Codes erforderlich sein.

Der von TETRA erzeugte C++-Code benutzt die **Bibliothek für reguläre Ausdrücke** von Dr. Maddock, der zu finden ist unter

http://www.boost.org/

Die Bibliothek ist konform mit der C++ Standard Library und ist unter fast allen aktuellen Betriebssystemen mit unterschiedlichen Compilern getestet worden. Genaue Angaben hierzu sind zu finden unter

http://www.boost.org/development/tests/trunk/developer/summary.html

Xerces-C++ ist ebenfalls portabel. Eine Liste der unterstützen Betriebssysteme und Compiler befindet sich unter:

http://xml.apache.org/xerces-c/

Für TETRA wurde Xerces-C++ mit Borlands CBuilder 6 angefertigt. Um einen Konflikt mit einer ebenfalls von Borland ausgelieferten dll zu vermeiden wurde der vorgeschlagene Name "XercesLib" in "**TTXercesLib**" geändert.

13.5 Lizenz

TextTransformer

Teil XIV

14 TetraComponents

Die freien TetraComponents erlauben es TextTransformer-Projekte innerhalb von Delphi oder CBuilder-Programmen zu benutzen. Sie werden mit der ebenfalls frei verf ügbaren tetra_engine.dll ausgeführt (interpretiert). Die Komponenten kapseln diese dll.

Die Komponenten erlauben u.a. einige Behandlungen von Parse-Ereignissen. Dabei werden Nummern für Token und Produktionen übermittelt (in OnAcceptToken und OnEnterProduction).

Diese enumerierten Werte werden per default in die Schablone "enums_pas.frm" geschrieben:

```
unit -->NameSpace;

interface

type

  EToken = (
  -->TokenEnums
  );

  EProduction = (
  -->ProductionEnums
  );


const

   Tokens := TStringList.Create;
-->TokenNames
  );

   Productions := TStringList.Create;
-->ProductionNames
  );

implementation

end.
```

Mit dem Pfeil "-->" und anschließendem Schlüsselwort sind die Positionen gekennzeichnet, an die der TextTransformer project-spezifischen Code einfügt.
Die Schablone kann in einem Editor bearbeitet werden.

TextTransformer

Teil XV

15 Meldungen

In diesem Kapitel werden einige Fehlermeldungen und Warnhinweise erläutert, die beim Parsen oder Ausführen eines Projekts auftreten können.
Treten solche Meldungen auf, so werden sie zunächst innerhalb des Fehlerfensters aufgelistet. Der vollständige Meldungstext kann sichtbar gemacht werden, indem der entsprechende Eintrag doppelt angeklickt wird. Die Meldung erscheint dann im zentralen Meldungsfenster. Wenn der Eintrag einmal angeklickt wird und dann die Taste **F1** gedrückt wird, erscheint der entsprechende Text aus dieser Hilfe.

15.1 Unbekanntes Symbol: "xxx"

Der Grammatiktest auf Vollständigkeit ist fehlgeschlagen.

15.2 "X" kann nicht in Terminalsymbole abgeleitet werden

Der Grammatiktest auf Terminalisierbarkeit ist fehlgeschlagen.

15.3 Zirkuläre Ableitung: "X" . "Y"

Der Grammatiktest auf Zirkularität ist fehlgeschlagen.

15.4 "X" ist löschbar

Beim LL(1)-Test wurde eine löschbare Struktur gefunden.

15.5 LL(1) Fehler: "X" ist Anfänger mehrerer Alternativen

Beim LL(1)-Test wurden Alternativen gefunden, die mit dem gleichen Token beginnen.
.

15.6 LL(1) Warnung: "X" ist Anfänger und Nachfolger löschbarer Strukturen

Ein möglicher wurde gefunden. LL(1)-Fehler

15.7 "X" ist ein SKIP-Knoten mit benachbarten Skip-Knoten

Der Grammatiktest auf eindeutige SKIP-Symbole ist fehlgeschlagen.

15.8 Löschbare Struktur in einer Wiederholung oder Option

Die Bedeutung einer **"(...)+"** oder **"(...)*"** Wiederholung oder einer **"(...)?"** Option ist nicht eindeutig, wenn sie eine löschbare Struktur enthält.

Beispiele:

```
( "a"? )+ "b"
```

Die Struktur "("a"?)+" könnte als löschbar interpretiert werden, so dass Auffinden von *b* im Text keinen Fehler ergäbe. Mit gleichem Recht könnte man sagen das direkte Fortschreiten zu *b* wäre falsch, da die Schleife mindestens einmal durchlaufen werden muss. Eine dritte Möglichkeit wäre vielleicht, vor dem Akzeptieren von b die Schleife einmal zu betreten und sogleich wieder zu verlassen.

```
( "a"* {{iCount++;}} )* "b"
```

Bei dieser Struktur wäre nicht klar, wie oft *iCount* bei mehreren Vorkommen von "a" inkrementiert würde.

15.9 Einschluss nicht gefunden

In den Projekteinstellungen oder in den lokalen Einstellungen einer Produktion ist eine nicht vorhandene Einschluss-Produktion gesetzt.

15.10 "X" wird in einer Vorausschau zirkulär verwendet

Diese Meldung erscheint, wenn eine Vorausschau nicht ausgeführt werden kann, weil sie zirkulär ist.

15.11 Konflikt mit einem Einschluss

Das in der Grammatik folgende Token ist auch der Anfang einer Einschluss-Produktion.

15.12 Es wurde kein passendes Nachfolge-Token gefunden

Der Versuch ein Nachfolger-Token zu ermitteln ist fehlgeschlagen. Sämtliche der in Frage kommenden Token passen nicht zum aktuellen Text.

15.13 Der restliche Text besteht aus bedeutungslosen Zeichen

Die Meldung: "Der restliche Text besteht aus bedeutungslosen Zeichen" weist darauf hin, dass am Ende des Eingabetextes noch Zeichen stehen, die die in den Optionen als auszulassende Zeichen bestimmt sind, und die daher nicht in den Ausgabetext übernommen wurden. Diese meldung ist ein bloßer Hinweis und bedeutet keinen Fehler.

15.14 SKIP-Token passt an aktueller Position

Diese Meldung erscheint im Log-Fenster, wenn eines der Token bis zu dessen Vorkommen der Text übersprungen werden sollten, bereits an der aktuellen Textposition vorhanden ist.

Beispiel:

```
SKIP "end"
```

wird angewandt auf den Text

```
"end ..."
```

SKIP-Token werden nur dann akzeptiert, wenn sich zwischen der aktuellen Position und dem Ort des Tokens Text befindet, der aus nicht auszulassenden Zeichen besteht:

```
"... end"
```

15.15 "SKIP ANY" ist nicht möglich

Mit SKIP kann nur zu einer explizit definierten Menge von Token gesprungen werden.

15.16 Passendes Token nicht in der Anfängermenge

Diese Meldung erscheint im Log-Fenster, wenn ein Token zwar auf den Text passen würde, es aber nicht ausgewählt wird, weil es nicht zur aktuellen Anfängermenge passt.

Beispiel

15.17 Passendes aber nicht akzeptiertes Token

Diese Meldung erscheint im Log-Fenster, wenn ein Token im Text erkannt wurde, dass gemäß der Grammatik nicht zulässig ist.

Beispiel:

```
Eingabe:   Text2Html.ttp
Regel:     SKIP? EOL
Meldung:   Passendes aber nicht akzeptiertes Token: ID ( 7 ), \w+
```

Hier wurde "Text2Html" als Bezeichner *ID* erkannt, obwohl das gesamte "Text2Html.ttp" hätte übersprungen werden sollen.

Als Ausweg kann entweder die Option *Globaler Scanner für reguläre Ausdrücke* abgeschaltet werden oder die Regel kann umformuliert werden:

```
Regel:     ( SKIP | ID )* EOL
```

15.18 Passende Vorausschau xxx kann nicht mit yyy beginnen

Diese Meldung erscheint, wenn eine Vorausschau erfolgreich war, aber das Token mit dem sie begann, nicht zur Anfängermenge des IF-Zweigs gehört. Das kann der Fall sein, wenn im Hauptparser ein Token erkannt wurde, dass nicht in der Anfängermenge der Vorausschau vorkommt, der gleiche Text aber dennoch auch von anderen Token in der Vorausschau erkannt wird.

15.19 Unerwartetes Symbol in ..

Ein Syntaxfehler innerhalb einer Regel ist aufgetreten. Ein Textstück kann nicht interpretiert werden. Entweder wurde ein nicht definiertes Wort verwendet oder ein zulässiges Schlüsselwort kann falsch geschrieben sein. Häufige Ursache sind auch falsch gesetzte Klammern oder Anführungszeichen. Möglicherweise fehlt ein erwarteter Parameter.

15.20 Unerwartete Methode, möglicherweise ...

Die Meldung:

Unerwartetes Methode (möglicherweise auch eine Klassenfunktion, die einen Wert zurückgibt)

erscheint, wenn eine Anweisung mit einem Bezeichner beginnt, auf den zunächst ein Punkt folgt und dann ein Symbol, das keine der möglichen Klassenmethoden bezeichnet. Z.B.

```
s.unknown
```

Eine falsche Verkettung von Methoden kann diese Meldung zur Folge habe. Z.B. ist clear eine Methode ohne Rückgabewert, an clear kann daher keine zweiter Methodenaufruf angehängt werden.

```
s.clear().clear();
```

Die Meldung erscheint auch dann, wenn versucht wurde eine Klassenmethode aufzurufen, die einen Wert zurückliefert, z.B. für den String s:

s.length();

Korrekt wäre:

```
int i = s.length();
```

In der C++-Syntax ist dies eine korrekte Anweisung. Sie ist jedoch sinnlos, da der Rückgabewert nicht verwendet wird. Hier wird deshalb mit der Fehlermeldung reagiert.

15.21 "X" erwartet

Gemäß der Grammatik hätte ein Token vom Typ "X" auf das zuletzt erkannte Token folgen müssen. Lautet z.B. die Regel für eine Anrede:

```
"Hallo" Name "!"
```

und steht im Text: "Hallo !",

erscheint diese Fehlermeldung, da nach "Hallo" ein Name erwartet wird.

15.22 Klammern benötigt

Die Warnmeldung: Klammern sind erforderlich, wenn die erste semantische Aktion eine Variablendeklaration für die ganze Produktion enthält, erscheint für Produktionen der Art

```
{-...-}
A | B ...
```

Oft enthält die erste semantische Aktion Deklarationen für Variablen, die in allen Alternativer verwendet werden sollen. Dann müssen die Alternativen aber in Klammern gesetzt werden. Z.B.:

```
{- str s; -}
(
  "a" | "b"
)
{-return s;-}
```

Ohne die Klammern

```
{- str s; -}
  "a"
| "b"
{-return s;-}
```

erhält man für {-return s;-} die Fehlermeldung

Unbekannter Bezeichner s

Implizit besteht dann folgende Klammerung:

```
(
  {- str s; -}
    "a"
)
|
(
  "b"
  {-return s;-}
)
```

15.23 Unvollständig geparst

Die Fehlermeldung "Unvollständig geparst" bezieht sich auf die Grammatik des Parsers. Diese wurde bis zum Ende des Eingabetextes nicht vollständig abgearbeitet. D.h. am Ende des Textes wären noch weiter Token zu erwarten gewesen.

15.24 Schließendes Anführungszeichen fehlt

Bei einem unmittelbar in der Produktion definierten literalen Token fehlt entweder das schließende Anführungszeichen oder es befindet sich nicht in derselben Zeile wie das öffnende.

15.25 Literale dürfen nicht leer sein

Leerstrings sind keine Token. Insbesondere müssen auch die unmittelbar in einer Produktion definierten literalen Token aus mindestens einem Zeichen bestehen.

15.26 Fortsetzung mit C++-Code erwartet

Diese Fehlermeldung erscheint, wenn die Klammer einer semantischen Aktion nicht geschlossen wurde. Vom Produktionen-Parser wird dies daran erkannt, dass eine neue Aktion beginnt. Z.B.:

```
{{
str s;
// fehlende schließende Klammer
( expression[s] )*
{{
return s;
}}
```

15.27 Leere Alternative

Der vollständige Fehlertext lautet:

> Leere Alternativen müssen mit einer semantischen Aktion verknüpft sein

Eine genaue Erklärung zu diesem Punkt wird auf der Seite über Alternativen innerhalb von Produktionen gegeben.

15.28 Fehler in der Parameterdeklaration

Eine Fehlermeldung der Art:

xxx: Fehler in der Parameterdeklaration "unexpected symbol" in: type_specifier_Alt0

tritt auf, wenn der Text in dem Parameterfeld der Regel "xxx" nicht korrekt geparst werden kann. Dies kann entweder an einem Schreibfehler des Parametertyps liegen oder daran, dass der Parametertyp dem Interpreter unbekannt ist, möglicherweise gar nicht interpretiert werden soll, sondern für den Export gedacht ist. (Parameter und {{...}})

15.29 Deklaration von Parametern stimmt nicht mit der Verwendung überein

Diese Fehlermeldung (oder Warnung s.u.) erscheint, wenn eine Produktion oder ein Token entweder mit einer falschen Anzahl von Parametern aufgerufen wird oder, wenn die Parameter-Typen nicht konvertibel sind.

Die Fehlermeldung erscheint beispielsweise im folgenden Fall. Die Parameterdeklaration der Produktion XXX sei:

```
Parameter: int xs
```

und der Aufruf laute:

```
{{str s = "Hallo";}}

XXX[s]  // falscher Parameter-Typ
```

Eine Variable vom Typ str ist nicht in einen "int"-Wert konvertierbar. Auch das Weglassen des Parameters:

```
XXX  // fehlender Parameter
```

führt zur gleichen Fehlermeldung.

Als Warnung erscheint diese Meldung, wenn die Produktion innerhalb einer Vorausschau verwendet

wird. In Vorausschau-Produktionen werden keine semantischen Aktionen durchgeführt und demnach auch Parameter nicht verwendet. In komplexen Projekten kann es aber vorkommen, dass die gleiche Produktion auch mit semantischen Aktionen verwendet wird, was u.U. erst bei Ausführung des Transformations-Programms zu einem Fehler führt.

Um sicher zu gehen, dass kein Fehler auftreten kann, können "Dummy"-Parameter eingesetzt werden.

15.30 Falsche Anzahl von (interpretierbaren) Argumenten

Ein Skript, für das eine bestimmte Anzahl von Parametern definiert wurde, wird mit einer anderen Anzahl aufgerufen.
Beispielsweise könnten für die Produktion *Name* zwei Parameter definiert sein:

```
Parameter : str sVorname, str sNachname
```

Wird sie dann aufgerufen mit:

```
Name["John"]
```

erscheint die Fehlermeldung.

Grund der Fehlermeldung ist möglicherweise auch, dass ein dem Aufruf übergebenes Argument nicht interpretierbar ist. Ist in den Projektoptionen eingestellt, dass nicht geklammerte Ausdrücke nur als exportierbar gelten, so ergibt folgender Code den Fehler:

```
Parameter : {= str sVorname, str sNachname =}

Aufruf : Name["John", "Smith"]
```

15.31 Nicht konstante Methode

Die Meldung:

Nicht konstante Methode *xxx* für konstantes Objekt aufgerufen *XXX*

erscheint, wenn das Objekt *XXX* als konstant deklariert wurde, die Methode *xxx* des Objekts, dieses aber modifizieren würde.

Beispiel:

In einer Funktion mit dem

```
Parameter :  const mstrstr& xm
```

soll der Cursor von xm auf den nächsten Wert gesetzt werden:

```
xm.gotoNext();
```

Da der Cursor und seine Position jedoch zu mstrstr "gehören", würde die Map durch das Weiterrücken des Cursors modifiziert. Der Parameter muss also als nicht konstant deklariert werden.

```
Parameter :  mstrstr& xm
```

15.32 Maximale Stack-Größe von"x" überschritten

Diese Meldung erscheint, wenn der interne Stack den in den Projektoptionen eingestellten Wert überschreitet.

Anm.: Der interne Stack ist dabei größer als der angezeigte da er sämtliche Verzweigungen enthält und nicht nur die Verzweigungen in Unterregeln.

15.33 Fehler beim Parsen der Parameter des Parseraufrufs:

Diese Meldung zeigt einen Fehler an der nur dann auftreten kann, wenn die Startregel des Parsers Parameter erfordert. Ist die der Fall, so müssen zur Erzeugung des Codes der Parserklasse, die Namen der Variablen aus der Deklaration der Parameter der Startregel extrahiert werden, ungeachted dessen, ob die Deklaration interpretierbar ist oder nicht.

Beispiel:

Heiße die Startregel "Startregel" und benötigt sie den Parameter: int xi, so wird folgender Code zum Aufruf des Startregel-Parsers erzeugt:

```
void CStartregelParser::Params(ctsr xtBegin, cts xtEnd, int xi)
{
  sps xState(xtBegin, xtEnd);
  Startregel(xState, xi);
}
```

Der Name des Parameters "xi" wurde hierzu aus der Deklaration "int xi" extrahiert. Träte hierbei ein Fehler aus, so erschiene die hier beschriebene Meldung. "Fehler beim Parsen der Parameter des Parseraufrufs: ..."

15.34 Es gibt mindestens einen Pfad, auf dem kein xxx-Wert zurückgegeben wird

Für die Produktion oder das Token ist ein Rückgabetyp *xxx* deklariert, aber es gibt eine Möglichkeit den Skripttext zu durchlaufen, bei der kein Wert zurückgegeben wird.
Beispielsweise könnte ein Tokenskript dazu dienen abhängig von einer erkannten Zahl verschiedene Strings zurückzuliefern:

```
str sDefault;
switch(xState.itg())
{
```

```
    case 1:
    return "eins";
    case 2:
    return "zwei";
    default:
    sDefault = xState.str();
  }
```

Im default-Fall wird hier kein Wert zurückgeliefert.

15.35 Erkanntes, aber nicht akzeptiertes Token

Falls die Ausführung eines TETRA-Programms durch einen Fehler abgebrochen wurde, kann es einen Hinweis auf ein erkanntes aber nicht akzeptiertes Token in der Log-Fenster geben. Vermutlich trat im Interpreter ein **Fehler** auf, bevor das erwartete Token akzeptiert werden konnte. Oder es wurde versehentlich eine **Return-Anweisung** vor die Ermittlung des nächsten Tokens gesetzt.

Beispiel:

```
ID
{{ return true; }}
NUMBER
```

Hier wird nach dem ersten Bezeichner **ID** eine Zahl **NUMBER** erwartet. **NUMBER** kann aber nicht erkannt werden, da die Produktion zuvor mit *return* verlassen wird.

Möglich ist auch, dass ist die Benutzung eines globalen Scanners für reguläre Ausdrücke aktiviert ist, obwohl zwei Token dieses Scanners miteinander in **Konflikt** stehen können.

Bestehe beispielsweise eine kleine Tabelle aus den beiden Spalten Version und Preis eines Produkts und seien diese definiert als

```
Version = \d\.\d\d
Price = \d\d?\.\d\d
```

d.h. eine Preisangabe kann eine oder zwei Vorkommastellen haben, eine Versionsangabe stets aber nur eine. Der Inhalt der Tabelle kann nun mit der Regel geparst werden:

```
Table = ( Version Price )*
```

Wird nun ein globaler Scanner benutzt, so ist nicht klar, ob eine Zahl mit nur einer Vorkommastelle durch *Price* oder durch *Version* erkannt wird. Befindet sich die Zahl in der ersten Spalte, erfolgt die Erkennung dennoch möglicherweise durch Price und das Parsen wird mit der Fehlermeldung abgebrochen, dass Price zwar passt, aber nicht akzeptiert wird.
Bei Verwendung lokaler Scanner ergibt sich hingegen keinerlei Problem, da je nachdem, ob sich die Zahl in der ersten oder zweiten Tabellenspalte befindet, jeweils nur auf eine Versionsnummer oder auf einen Preis getestet wird.

15.36 BREAK außerhalb einer Schleife

Das **BREAK**-Symbol kann nur innerhalb einer Schleife (...)* oder (...)+ und in der gleichen Produktion wie die Schleife selbst verwendet werden. An allen anderen Positionen erfolgt die entsprechende Fehlermeldung beim Parsen.

15.37 Stillstand

Trotz Parsens, wurde die Position im Eingabetext nicht verändert.

Ein Token, dass auf einen Text passt, muss mindestens ein Textzeichen abdecken. Ein Token, wie

```
Token A:: = a*

Produktion ::= A
```

würde an jeder beliebigen TextPosition passen, da das Zeichen 'a' dort mindesten null mal vorkommt. Eine Regel wie (A)* würde damit eine Endlosschleife verursachen. Ein optionales Vorkommen einer Kette von 'a'-Zeichen sollte daher folgendermaßen formuliert werden:

```
Token A:: =  a+

Produktion =  ( A )?
```

15.38 Stillstand bei der Vorausschau

Bei der Vorausschau (look ahead) besteht die Gefahr eines unendlichen Regresses, wenn am Anfang einer Produktion diese Produktion selbst in einer IF-Produktion zur Vorausschau verwendet wird. Beispiel:

```
ab   =  IF( ab ) ...
```

Bevor eine Vorausschau mit einer Produktion gestartet wird, wird daher geprüft, ob der Parser sich schon in einer Vorausschau befindet. Falls dies der Fall ist, und falls es in der Vorausschau bisher noch keinen Fortschritt im Text gegeben hat, wird der Parser gestoppt und es erscheint die Fehlermeldung:

Stillstand bei der Vorausschau

15.39 Unbekannter Bezeichner : xxx

Die Meldung "Unbekannter Bezeichner : xxx" ist durch einen Fehler im Interpretercode verursacht. Eine Variable wurde verwendet, die nicht zuvor deklariert wurde. Die Ursache dieser Meldung kann bisweilen versteckt sein (s.u.).
Die Meldung kann auch als Folge des Auftretens eines anderen, vorhergehenden Fehlers erfolgen.

Beispiel

Die Zuweisung

```
s = "Hallo";
```

erzeugt die Fehlermeldung "Unbekannter Bezeichner : s", wenn die Variable s nicht zuvor deklariert wurde. Richtig wäre daher:

```
str s;
s = "Hallo";
```

oder in einer Anweisung zusammengefasst:

```
str s = "Hallo";
```

Für Interpreter unsichtbare Deklaration in Exportcode

Die Fehlermeldung erscheint z.B auch, wenn die Deklaration einer Variablen in den Klammern "{_" und "_} erfolgt:

```
{_str sComment;_}
```

die Variable anschließend aber im Interpretercode auftaucht:

```
(
 Comment[sComment]
)+
```

sComment wird als unbekannter Bezeichner reklamiert, wenn in den Projektoptionen für Parameter und "{{...}}" **interpretierbar** aktiviert ist.

versteckte Sichtbarkeitsbereiche

Eine einmal deklarierte Variable existiert, solange ihr Sichtbarkeitsbereich besteht. Ein Sichtbarkeitsbereich ist der gesamte Text einer Produktion/Token oder der Bereich der durch das Klammerpaar '{' und '}' begrenzt ist oder er kann indirekt durch Alternativen definiert sein.
Lautet der Text einer Produktion z.B.:

```
if( xi == 1 )
{
  str s = "eins";
  out << s;
}
```

```
s = "zwei"  // Fehler
```

so kann dem str s in den Zeilen, die der schließenden Klammer '}' folgen, nichts mehr zugewiesen werden, da er nicht mehr existiert.

Achtung: Alternativen definieren Sichtbarkeitsbereiche, die nicht unmittelbar zu erkennen sind.

Beispiel

Die folgende Produktion verursacht die Fehlermeldung "Unbekannter Bezeichener : s", obwohl der str s korrekt deklariert scheint:

```
{{str s; }}
 "a"
| "b"
  print[s]
```

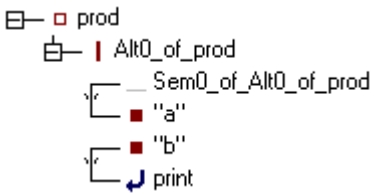
Tatsächlich werden durch die Alternative versteckte Sichtbarkeitsbereiche eingeführt.

```
{{str s; }}
 "a"
```

und

```
  "b"
  print[s]
```

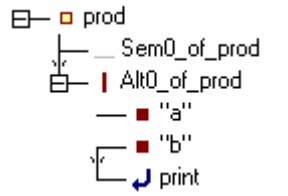
stehen jeweils in einem eigenen Bereich. Deutlich wird dies in der Baumdarstellung der Produktion:



Die semantische Aktion, die den str deklariert, wird unmittelbar vor der Erkennung von "a" ausgeführt und nicht vor der gesamten Alternative. Um dieses Ziel zu erreichen, muss die Alternative geklammert werden:

```
{{str s; }}
(
 "a"
| "b"
  print[s]
)
```

Die Baumdarstellung sieht nun folgendermaßen aus:



Die Deklaration steht nun vor der gesamten Alternative und auf die Variable s kann in der gesamten Produktion zugegriffen werden. Der Fehler ist behoben.

15.40 Konvertierung von "xxx" nach "yyy" ist nicht möglich

Die Meldung "Konvertierung von "xxx" nach "yyy" ist nicht möglich" ist durch einen Fehler im Interpretercode verursacht. Es wurde versucht einer Variablen einen Wert zuzuweisen, der nicht in den Variablentyp konvertierbar ist.

Beispiel

```
int i = "Hallo";
```

Ein String kann nicht in eine Zahl umgewandelt werden.

15.41 Es wurde kein Rückgabetyp deklariert

Im Interpretercode wird ein Wert mit *return* zurückgegeben, das Feld des Rückgabetyps des Scripts ist jedoch leer.

15.42 "X" kann nicht angewandt werden auf "Y"

Dieser Fehler wird vom Interpreter erzeugt, wenn versucht wird auf den Typ "Y" eine Operation "X" anzuwenden, die für diesen Typ nicht definiert ist.
Z.B.:

```
str s;
s++;
```

15.43 break- oder continue-Anweisung an ungültiger Position

Eine *break*- oder eine *continue*-Anweisung kann nur innerhalb der Schleife einer for-, while-oder do-while-Anweisung verwendet werden. *break* findet außerdem noch Verwendung in switch -Anweisungen. An allen anderen Stellen ist die Verwendung von *break* und *continue* unzulässig.

Dieser Fehler wird erst bei der Ausführung eines Programms erkannt.

15.44 verbotene Übergangsaktion

Nur solche Übergangsaktionen sind erlaubt, die den Zustand des Parsers vor und nach Verwendung der Produktion zur Vorausschau unverändert lassen. Die der Kette der Übergangsaktionen enthält mindestens eine Funktion, für die dies nicht gewährleistet ist.

15.45 Der Typ der Funktion xxx passt nicht zur Funktionstabelle

Sämtliche Funktionen einer Funktionstabelle müssen den gleichen Rückgabetyp und die gleiche Anzahl und Typen von Parametern haben.

15.46 Für Funktions-Tabelle ist keine Default-Funktion definiert

Jede Funktionstabelle muß eine Default-Funktion enthalten. Die Default-Funktion wird mit einem Leerstring als Schlüssel in die Tabelle eingefügt. Sie wird automatisch für diejenigen Knoten aufgerufen, die per *visit* besucht werden, jedoch keinen Label-Wert haben, der als Schlüssel in der Funktionstabelle vorkommt.

15.47 In einem const Parser muss die entsprechende State-Methode aufgerufen werden

Falls die const-Option in den Projektoptionen gesetzt ist, können die Plugin-Methoden nicht direkt aufgerufen werden, sondern müssen als Methode von xState aufgerufen werden.

Beispiel:

statt

```
str s = AddToken("Print". "USERFUNCTION");
```

muss es heißen

```
str s = xState.AddToken("Print". "USERFUNCTION");
```

15.48 Unterausdrücke (> 0) sind im LA-Puffer nicht gespeichert

Auf die durch Klammern markierten Unterausdrücke kann in den semantischen Aktionen nicht zugegriffen werden, wenn die Vorausschau-Token gepuffert werden.

15.49 Produktion kann nicht als Einschluss verwendet

Eine Produktion kann nicht als Einschluss verwendet werden, wenn sie von einer anderen Produktion aufgerufen wird (es sei denn rekursiv innerhalb des Einschlusses selbst).

15.50 Einschluss mit Parametern

Eine Produktion, die einen Einschluss beginnt darf keine Parameter haben. Eine Grammatik kann an beliebiger Stelle durch einen Einschluss "unterbrochen" werden. Daher gibt es keinen Ort, an dem ein Parameter übergeben werden könnte. Um Informationen zwischen dem Hauptparser und einem Einschluss auszutauschen müssen daher Klassenvariablen verwendet werden.

Bei der Erzeugung von Quellcode für const-Parser empfiehlt es sich, Variablen des Plugins zum Informationstransport zu verwenden.

15.51 Einschlüsse funktionieren nicht mit einem LA-Puffer

Einschlüsse können nicht verwendet werden, wenn die Vorausschau-Token gepuffert werden.

15.52 State-Parameter erforderlich

Diese Meldung erscheint nur dann, wenn in den Projektoptionen auf der Register-Seite *Warnungen/Fehler* die Checkbox xState-Parameter für Klassenfunktionen mit einem Häkchen versehen ist. *xState* muss dann als erster Parameter in den Aufrufen von selbssdefinierten Klassenmethoden stehen. Nur so ist der Interpreter-Code auch als C++-Code exportierbar.

15.53 Vom Benutzer programmierte Fehlerausgabe:

Meldungen die mit dem Text:

> Vom Benutzer programmierte Fehlerausgabe:

eingeleitet sind, stammen aus einem Aufruf im Interpreter von:

```
throw CTT_Error("Fehlermeldung");
```

15.54 Zweig kann nicht eingefügt werden

Diese Meldung erscheint im Log-Fenster und beendet die Ausführung eines Programms, in dem versucht wurde ein node-Objekt, das bereits Teil eines Baums ist erneut einzufügen.

15.55 Fehler in Token

In der Definition eines regulären Ausdrucks auf der Tokenseite ist ein Fehler aufgetreten.

15.56 Passt auf Leerstring

Token dürfen nicht auf einen leeren Text passen -> Stillstand

15.57 Token ist als String und als Token mit Aktion definiert

Die Warnung:

"xxx" ist als String und als Token mit Aktion definiert

erfolgt, wenn ein Token auf der Produktionen-Seite als String definiert wird, für den es auf der Token-Seite bereits ein Token gibt, das mit einer Aktion verbunden ist. Es ist prinzipiell möglich in einem Projekt beide Token zu verwenden, wenn sie keine unmittelbaren Alternativen sind. Ob dies der Fall ist, ist jedoch oft schwer zu erkennen und es ist daher sehr zu empfehlen sich auf eine der Definitionen zu beschränken. Andernfalls könnte es zu unbeabsichtigter (Nicht-)Ausführung der Token-Aktion kommen.
Wenn mit dem auf der Token-Seite definierten Token keine Aktion verbunden ist, wird keine Warnung ausgegeben.

15.58 boost::regex-Fehler

Mit dieser Meldung im Log-Fenster wird die Ausführung eines TextTransformer-Programms abgebrochen, wenn ein Fehler in der Klasse für reguläre Ausdrücke der boost-Bibliothek auftritt. Die Ursache ist vermutlich ein zu komplexer Ausdruck. Abhilfe kann geschaffen werden, indem das Friedl-Schema angewandt wird. Nötigenfalls können auch literale Teile des Ausdrucks abgespalten und der gesamte Ausdruck in eine Produktion verlagert werden.

15.59 Überlappende Systeme

Produktionen können im TextTransformer auf vierfache Weise verwendet werden. Wird eine bestimmte Produktion auf mehrfache Weise verwendet, so ist die Menge der Token auf die in ihr getestet wird, durch die zuerst geparste Verwendungsweise bestimmt.

15.60 Token-Aktion oder Klassenfunktion kann nicht exportiert werden

Eine Token-Aktion oder Klassenfunktion kann nicht exportiert werden, wenn sie interpretierbare Code-Teile besitzt, zwischen denen ein Code-Teil eingebettet ist, der allein für den Export bestimmt ist.

Beispiel:

```
{= ... =}   // interpretierbarer und exportierbarer Code
{_ ... _}   // allein für den Export bestimmt
{= ... =}   // interpretierbarer und exportierbarer Code
```

15.61 Hier darf nur Initialisierungscode stehen

In dem Textfeld auf der Element-Seite dürfen für Variablen nur einfache Initialisierungen stehen. Anderer Code erzeugt die Fehlermeldung: Hier darf nur Initialisierungscode stehen.

15.62 Parameter und lokale Variablen dürfen in einer Vorausschau-Produktion nicht verwendet werden!

Diese Fehlermeldung erscheint, wenn das Parsen mit einer Produktion durch semantischen Code gesteuert wird, der auf eine lokal deklarierte Variable oder einen Parameter zugreift, die Produktion aber zugleich direkt oder indirekt zur Vorausschau verwendet wird. Dies ist nicht zulässig, weil in Vorausschau-Produktionen zur Vermeidung von Seiteneffekten wie der Verdopplung einer Textausgabe, sämtlicher semantischer Code ignoriert wird, soweit er nicht unmittelbar zur Bedingung einer IF- oder WHILE-Struktur gehört.

Beispiel:

```
IF( Prod() )
  ...
```

Die Produktion *Prod* wird in einer IF-Struktur als Vorausschau-Produktion eingesetzt. Sie enthält aber eine WHILE-Struktur, deren Auswertung auf die lokal deklarierte Variablen *b* zugreift.

```
Prod ::=
{{
bool b = true;
}}

WHILE ( b )
... {{b = false;}}
END
```

Die Verwendung von Klassenvariablen wird in derartigen Fällen geduldet, aber er ist gefährlich. So könne man eine Klassenvariable *m_b* auf der Element-Seite deklarieren und der Code könnte geändert werden zu:

```
WHILE ( m_b )
... {{b = false;}}
END
```

Aber innerhalb einer Vorausschau würde dieser Code eine Endlosschleife erzeugen, weil in der , but

it is dangerous So a member variable m_b could be declared on the element page and the code could be changed to:

```
WHILE ( m_b )
... {{b = false;}}
END
```

But inside of the look-ahead this would result in an endless loop, because in a look-ahead "b = false" is not executed. "b = false" Nicht ausgeführt wird.. Auch kann ein solcher Gebrauch im generierten Code zu Verletzung von const-Bedingungen führen.

15.63 Kodierung kann nicht in das Ausgabefenster der IDE geschrieben werden

Diese Meldung erscheint im Ausgabefenster, wenn die Kodierung eines XML-Dokuments dort nicht dargestellt werden kann.

15.64 An invalid or illegal XML character is specified

Diese Meldung erscheint im Log-Fenster in englischer Sprache, weil sie von xerces erzeugt wurde. Sie besagt, dass zur Definition einer *dnode* ein Zeichen verwendet wurde, dass nicht verwendet werden darf.

Beispiel:

```
nLine.add(",",   xState.str()); //  =>  Laufzeitfehler
```

15.65 TextTransformer nicht registriert

Index-Operationen sind nur in der registrierten Version des TextTransformers zulässig.
Index-Operationen sind Operationen bei denen über einen Index auf einzelne Elemente einer Struktur zugegriffen wird. Beispiele hiefür sind Zugriffe auf die Elemente des mstrstr-Containers oder auch auf die Unterausdrücke des zuletzt erkannten Tokens. Was genau in der freien Version erlaubt ist, ist dieser Hilfe zu entnehmen.

15.66 Internal error : ...

Fehlermeldungen, die mit den Worten "Internal error" eingeleitet werden, sollten sie hoffentlich nicht zu sehen bekommen. Sie werden erzeugt, wenn programminterne Bedingungen nicht erfüllt sind, zeigen also einen Programmfehler des TextTransformers selbst an. Die Meldungstexte, die auf "Internal error . " folgen sind zumeist in englischer Sprache verfasst.
Bitte teilen sie mir das Auftreten solcher Fehler, samt der Umstände, die zu ihm führten mit. Benutzen sie für eine solche Mitteilung möglichst das Feedback-Formular. Damit helfen sie, die

Software zu verbessern.

15.67 Keine Hilfe

Es ist leider nicht vollständig ausgeschlossen, dass nach dem Drücken der F1-Taste kein Hilfsfenster erscheint, oder dass dieses Fenster erscheint, oder dass gar eine Fehlermeldung angezeigt wird, die **Hilfedatei sei korrupt**. Seien Sie unbesorgt: die Hilfedatei ist völlig in Ordnung. Das Windows-Hilfe-System erzeugt diese Meldung automatisch, wenn es einen Hilfeaufruf nicht richtig verarbeiten kann. Bitte unterstützen Sie die Entwicklung des TextTransformers und teilen Sie mit, unter welchen Umständen bei Ihnen das Hilfesystem versagt.

TextTransformer

Teil XVI

16 Referenzen

16.1 Referenzen

Die Entstehung des TextTransformers wäre nicht denkbar ohne die im folgenden genannten Veröffentlichungen. All den genannten Autoren bin ich zu Dank verpflichtet.

TETRA basiert auf dem Konzept für den TopDown-Compiler-Compiler **Coco von P. Rechenberg und H. Mössenböck**, das sie in Ihrem gemeinsam verfassten Buch: Ein Compiler-Generator für Mikrocomputer, Carl Hanser Verlag München Wien 1988 dargelegt haben.

H. Mössenböck (ETH Zürich) hat eine ursprüngliche Version von Coco in Oberon-2 verfasst, die dann von Marc Brandis (ETH Zürich) und Pat Terry (Rhodes University, Grahamstown, South Africa) nach Modula-2 portiert wurde. Schließlich hat Frankie Arzu (Universidad del Valle, Guatemala, Central America) eine Version von Coco in C veröffentlicht.
Die aktuellen Versionen von Coco bzw. Coco/R, die auch Code für andere Zielsprachen als C++ produzieren können, sind zu finden unter:

http://www.ssw.uni-linz.ac.at/Reserach/Projects/#Coco

Der TextTransformer macht Gebrauch von einigen **boost Bibliotheken**.

http://www.boost.org

Diese Bibliotheken sind auch erforderlich, um den Code zu kompilieren, der vom TextTransformer erzeugt wird.

Der Quellcode für **regulären Ausdrücke** stammt **von Dr John Maddock**:

Die f**ormat library von Samuel Krempp** wird im Interpreter zur möglichen Formatierung des Ausgabetextes verwendet.

Auf der **filesystem library von Beman Dawes** beruhen die portablen Funktionen zur Pfad- und dateibehandlung.

Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization
obtaining a copy of the software and accompanying documentation covered by
this license (the "Software") to use, reproduce, display, distribute,
execute, and transmit the Software, and to prepare derivative works of the
Software, and to permit third-parties to whom the Software is furnished to
do so, all subject to the following:

The copyright notices in the Software and this entire statement, including
the above license grant, this restriction and the following disclaimer,
must be included in all copies of the Software, in whole or in part, and
all derivative works of the Software, unless such copies or derivative
works are solely in the form of machine-executable object code generated by
a source language processor.

Der Parser für den Interpreter basiert auf einer **ANTLR**-Grammatik die zu finden ist unter:

http://www.antlr.org/grammars/cpp

```
/*
 * PUBLIC DOMAIN PCCTS-BASED C++ GRAMMAR (cplusplus.g, stat.g, expr.g)
 *
 * Authors: Sumana Srinivasan, NeXT Inc.;           sumana_srinivasan@next.com
 *          Terence Parr, Parr Research Corporation; parrt@parr-research.com
 *          Russell Quong, Purdue University;       quong@ecn.purdue.edu
 *
 * VERSION 1.2
 *
 * SOFTWARE RIGHTS
 *
 * This file is a part of the ANTLR-based C++ grammar and is free
 * software.  We do not reserve any LEGAL rights to its use or
 * distribution, but you may NOT claim ownership or authorship of this
 * grammar or support code.  An individual or company may otherwise do
 * whatever they wish with the grammar distributed herewith including the
 * incorporation of the grammar or the output generated by ANTLR into
 * commerical software.  You may redistribute in source or binary form
 * without payment of royalties to us as long as this header remains
 * in all source distributions.
 *
 * We encourage users to develop parsers/tools using this grammar.
 * In return, we ask that credit is given to us for developing this
 * grammar.  By "credit", we mean that if you incorporate our grammar or
 * the generated code into one of your programs (commercial product,
 * research project, or otherwise) that you acknowledge this fact in the
 * documentation, research report, etc....  In addition, you should say nice
 * things about us at every opportunity.
 *
 * As long as these guidelines are kept, we expect to continue enhancing
 * this grammar.  Feel free to send us enhancements, fixes, bug reports,
 * suggestions, or general words of encouragement at parrt@parr-research.com.
 *
 * NeXT Computer Inc.
 * 900 Chesapeake Dr.
 * Redwood City, CA 94555
 * 12/02/1994
```

```
*
* Restructured for public consumption by Terence Parr late February, 1995.
*
* DISCLAIMER: we make no guarantees that this grammar works, makes sense,
*           or can be used to do anything useful.
*/
```

Der Funktionstabellen-Assistent wurde mit Hilfe der Open Source Wizard-Komponente von William Yu Wei erstellt, die Teil der Jedi-Vcl ist:

http://homepages.borland.com/jedi/jvcl/
http://sourceforge.net/projects/jvcl

Xerces-C++ unterliegt der Apache Software License, Version 2.0.

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.

Die "ATBinHex" Komponente von Alexey Torgashin wurde für den Betrachter der Quelldateien verwendet.

http://atorg.net.ru/delphi/atbinhex.htm

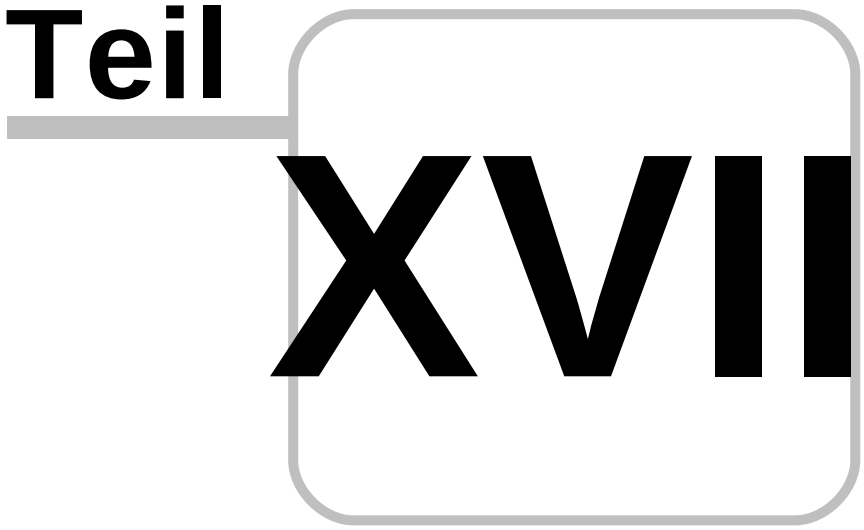
Die Komponente wurde um die Möglichkeit erweitert, Haltepunkte zu setzen. Die erweiterte Komponete gibt es bei

http://www.texttransformer.org

ATBinHex unterliegt der MOZILLA PUBLIC LICENSE Version 1.1, die dort beigefügt ist.

Mein besonderer Dank gilt ferner **Dr. Hans-Peter Diettrich**, der mir einige wertvolle Hinweise zu Verbesserung das Programmes gab und **Andreas Busch**, der mir geholfen hat den TextTransformer mehrbenutzerfähig zu machen.

TextTransformer



17 Glossar

17.1 Anfängermenge

Jeder Knoten im TETRA-Syntaxbaum und jedes Symbol der Grammatik ist durch zwei Mengen von Token charakterisiert: die Menge seiner Anfänger und die Menge seiner Nachfolger. Die Anfängermenge ist bei der durchgeführten TopDown-Analyse eines Textes von besonderer Wichtigkeit, da sie den Fortgang der Analyse bestimmt.

Die **Anfängermenge** eines Knotens enthält alle diejenigen Token, deren Erkennung im Text den Parser veranlassen, gerade diesen Knoten als nächsten zu wählen und nicht einen anderen. Passt der aktuelle Text zu keinem der Token der Anfängermenge des Knotens, so wird derjenige alternative Knoten gewählt, dessen Anfängermenge ein Token enthält, zu dem der Text passt. Gibt es keine derartige Alternative, so wird das Parsen des Eingabetextes mit einer Fehlermeldung abgebrochen. Um zwischen Alternativen eindeutig entscheiden zu können, müssen die Anfängermengen der Alternativen disjunkt sein, d.h. es darf kein Token geben, das in mehr als einer der alternativen Anfängermengen enthalten ist.
Die Anfängermengen entscheiden somit über den Fortgang der Textanalyse und in der Berechnung dieser Mengen besteht die zentrale Leistung des TextTransformers. Zur Berechnung dieser Mengen müssen mehrere Fälle unterschieden werden.

Der einfachste Fall ist der, bei dem ein ***Knoten ein Terminalsymbol*** *repräsentiert*. Hier besteht die Anfängermenge genau aus diesem Terminalsymbol selbst.

Beispiel:

Die Anfängermenge von "TETRA" ist die Menge, die als einziges Element das Token "TETRA" enthält.

Die Anfängermenge einer ***Kette von Symbolen*** ist die Anfängermenge ihres ersten Gliedes (sofern es nicht löschbar ist s.u.)

Beispiel:

Die Anfängermenge von "TETRA" "steht" "für" "Texttransformer" ... ist die Menge, die als einziges Element das Token "TETRA" enthält.

Mehr Elemente umfassen Anfängermengen von ***Knoten, die Alternativen enthalten***. Die Anfängermenge eines solchen Knotens besteht aus der Vereinigung aller Anfängermengen der einzelnen Alternativen.

Beispiel:

Die Anfängermenge von ("TETRA" ... | "Texttransformer" ...) ist die Menge der Token "TETRA" und "Texttransformer".

Am schwierigsten ist die Ermittlung von Anfängermengen ***löschbarer Knoten***. Das sind Optionen "(...)? und optionale Wiederholungen "(...)*". Zunächst sind Anfänger dieser Knoten alle Token, mit denen die alternativen Ketten innerhalb der optionalen Struktur beginnen. Für:

("sehr"+ | "super")? "gut"

ist dies die Menge aus den Token "sehr" und "super". Wegen der Optionalität des Knotens soll es aber nicht zum Abbruch des Parsens kommen, wenn an der aktuellen Textposition weder das Token "sehr" noch "super" zu finden sind. Das Parsen soll mit dem nachfolgenden Knoten fortgesetzt werden. Um beim aktuellen löschbaren Knoten bereits entscheiden zu können, ob das Parsen fortgesetzt werden kann muss die bisherige Anfängermenge {"sehr", "super"} mit der Nachfolgermenge vereinigt werden. (Dabei müssen die zu vereinigenden Mengen untereinander wiederum disjunkt (s.o.) sein.) Die Anfängermenge der Beispielsoption ist also: {"sehr", "super", "gut"}. Die Beispielsoption mag nun die Produktion *attribut1* bilden:

attribut1 = ("sehr" ... | "super" ...)? "gut"

Ein zweites Attribut sei schlicht:

attribut2 = "cool"

und eine Produktion laute:

urteil = "Der TextTransformer funktioniert" (*attribut1* | *attribut2*)

Wird mit der *urteil*-Produktion der Text:

"Der TextTransformer funktioniert gut"

analysiert, so wird zunächst "Der TextTransformer funktioniert" erkannt. Um entscheiden zu können, ob *attribut1* oder *attribut2* folgt, wird untersucht, ob der anschließende Text: " gut" überhaupt zu einem der in der Grammatik vorkommenden Token: {"sehr", "super", "gut", "cool"} passt. Er passt zu "gut" und "gut" gehört zur Anfängermenge von *attribut1* nicht aber zu der von *attribut2*. Somit ist klar, dass das erste Attribut zutrifft.

Die **Nachfolgermenge** ist die Menge derjenigen Token, die auf den Knoten folgen können

Semantische Aktionen werden innerhalb des Syntaxbaums auch als Knoten dargestellt. Bei ihnen handelt es sich um löschbare Knoten. Ihre Anfängermenge ist genau ihre Nachfolgermenge.

17.2 ASCII/ANSI-Zeichensatz

ASCII ist ein Akronym für "American Standard Code for Information Interchange". Der ASCII-Zeichensatz ist eine Zuordnung von Zahlen zu den Zeichen des US-amerikanischen Alphabets, den Ziffern und Interpunktionszeichen. Darüberhinaus enthält der ASCII-Zeichensatz sogenannte Steuerzeichen.

Im erweiterten ASCII-Zeichensatz können neben den US-amerikanischen Buchstaben auch Zeichen anderer Sprachen dargestellt werden (z.B. die deutschen Umlaute).

Neben dem Ascii-Zeichensatz gibt es auch andere Zeichensätze. Windows bevorzugt den ANSI-Zeichensatz (genauer: Windows-1252). Seit einigen Jahren gibt es den Unicode-Zeichensatz, der fast alle geschriebenen Sprachen der Welt wiedergeben kann.

Beispiele:

Das große 'A' hat den ASCII-Wert 65. Der ASCII-Code der Leertaste ist 32.

Die gesamte Tabelle ist über *Hilfe->ASCII-Tabelle* anzusehen.

Achtung: die Ascii-Nummer einer Ziffer ist nicht identisch mit ihrem numerischen Wert. Die Ziffer '2' beispielsweise hat den Ascii-Code 50.

17.3 Backtracking

Wenn bei der Analyse eines Textes ein Zustand eintritt, bei dem die gegenwärtige Information nicht ausreicht um mit Sicherheit zwischen mehreren Möglichkeiten des Fortgangs zu entscheiden, so müssen die einzelnen Möglichkeiten durch eine weitere Vorausschau im Text getestet werden. Schlägt ein solcher Test fehl, so muss wieder zu dem Ausgangszustand zurückgesetzt werden um die nächste Möglichkeit zu testen. Die Rückkehr in den Zustand von dem her die verschiedenen Alternativen getestet werden nennt man Backtracking.

17.4 Binärdatei

.
Im Unterschied zu einer Textdatei werden die Bytes einer Binärdateien nicht unbedingt als Textzeichen interpretiert. Solche Dateien werden durch spezielle Algorithmen z.B. als Bilder oder Töne interpretiert und dargestellt. Auch die Dokumente von Textverarbeitungen sind Binärdateien deren Formatierungsinformationen nur durch komplizierter Algorithmen zu entschlüsseln sind.

In Binärdateien können Bytes mit dem Wert 0 vorkommen. In Textdateien ist dies nicht erlaubt. Dieser Umstand wird von dem Beispielsprojekt *BinaryCheck* ausgenutzt.

17.5 Compiler

Ein Compiler ist ein Programm, dass Programme einer höheren Programmiersprache in Programme einer maschinennahen Sprache übersetzt.
Im weiteren Sinne kann jedes Programm. dass eine Programmiersprache in eine andere übersetzt (kompiliert) als Compiler bezeichnet werden. In diesem Sinne ist auch der TextTransformer ein Compiler, da er die Skripte eines Projekts zu einem Transformationsprogramm kompiliert.

17.6 Debuggen

Debuggen ist der Vorgang des Auffindens und Behebens von Programmfehlern (Bugs).

17.7 Escape-Sequenzen

Escape-Sequenzen können zur Repräsentation nicht darstellbarer Zeichen. Eine Escape-Sequenz beginnt stets mit dem Backslash. Z..B. steht \n für das Zeichen "Zeilenvorschub".

Ein Backslash in Kombination mit oktalen oder hexadezimalen Zahlen repräsentiert den ASCII- oder Steuercode, der diesem Wert entspricht, z.B. '\03' für Ctrl-C oder '\x3F' für das Fragezeichen. Eine Escape-Sequenz kann einen String mit bis zu drei oktalen oder beliebig vielen hexadezimalen Ziffern umfassen, vorausgesetzt, der Wert liegt im zulässigen Bereich des Datentyps char (0 bis 0xff). (Bei der Definition von regulären Ausdrücken werden nur zwei hexadezimale Zeichen ausgewertet.) Größere Zahlen liefern den Compilerfehler "value is too large". Die Oktalzahl \777 ist zum Beispiel größer als der zulässige Höchstwert \377 und ergibt somit einen Fehler. Das erste nicht-oktale oder nicht-hexadezimale Zeichen in einer oktalen oder hexadezimalen Escape-Sequenz markiert das Ende der Sequenz.

Sehen Sie dieses Beispiel:

```
out  <<  "\x0072.1A rudimentäres  Betriebssystem";
```

Diese Zeile soll als \x007 und "2.1A rudimentäres Betriebssystem" interpretiert werden. Tatsächlich wird \x0072 jedoch als hexadezimale Zahl und ".1A rudimentäres Betriebssystem" als literaler String interpretiert.

Zur Umgehung derartiger Probleme sollte die Programmzeile folgendermaßen neu geschrieben werden:

```
out  <<  "\x007" "2.1A rudimentäres  Betriebssystem";
```

Mehrdeutigkeiten entstehen auch, wenn auf eine oktale Escape-Sequenz eine nicht-oktale Ziffer folgt. Da 8 und 9 unzulässige oktale Ziffern sind, würde z.B. die Konstante \258 als Zwei-Zeichen-Konstante interpretiert, die sich aus den zwei Zeichen \25 und 8 zusammensetzt.

In der folgenden Tabelle sind alle Escape-Sequenzen zusammengefasst, die sowohl in literalen Token als auch in regulären Ausdrücken verwendet werden können. Für reguläre Ausdrücke gibt es noch weitere Escape-Sequenzen.

Hinweis: Sie müssen den (typischerweise in Pfadnamen verwendeten) Backslash verdoppeln (also "\\" angeben), um ihn als ANSI-Backslash gelten zu lassen.

Sequenz	Wert	Zeichen	Bedeutung
\a	0x07	BEL	Alarmton
\b	0x08	BS	Backspace
\f	0x0C	FF	Seitenvorschub
\n	0x0A	LF	Zeilenvorschub
\r	0x0D	CR	Wagenrücklauf
\t	0x09	HT	Tabulator (horizontal)

\v	0x0B	VT	Tabulator (vertikal)
\\	0x5c	\	Backslash (umgekehrter Schrägstrich)
\'	0x27	'	Einfaches Anführungszeichen (Apostroph)
\"	0x22	"	Doppeltes Anführungszeichen
\?	0x3F	?	Fragezeichen
\O		beliebig	O = ein String mit 1 bis 3 Oktalziffern
\xH		beliebig	H = ein String mit Hex-Ziffern
\XH		beliebig	H = ein String mit Hex-Ziffern

17.8 deterministisch

Grammatiken heißen deterministisch, wenn sie so aufgebaut sind, dass stets mit Sicherheit feststeht wie die Analyse eines Textes anhand dieser Grammatik fortzusetzen ist. Bei deterministischen Grammatiken ist also kein Backtracking erforderlich.

17.9 Friedl-Schema

Als Friedl-Schema wird hier ein Muster bezeichnet, das J.E.F. Friedl in seinem Buch: *Reguläre Ausdrücke*, formuliert hat, um *ewiges Matching* zu vermeiden.

Friedl nennt als besonders unglückliches Beispiel für *ewiges Matching*:

```
"(\\.|[^"\\\r\n]+)*"
```

Dieser Ausdruck besagt, dass ein String aus den beiden äußeren Anführungszeichen besteht und im Innern aus einer Wiederholung. Wiederholt wird die Alternative eines Backslashs gefolgt von einem weiteren Zeichen (so wird sichergestellt, das der Backslash nicht unmittelbar vor dem schließenden Anführungszeichen steht) und einer Folge von Zeichen, die weder Zeilenumbruchszeichen noch ein Anführungszeichen (das den String beenden würde) sind.
Das Problem an diesem Ausdruck ist, dass Zeichenfolgen vor einem Backslash auf mehrere Weisen interpretiert werden können und dass (bei einem POSIX-NFA Regex, wie er im TextTransformer verwendet wird) stets alle Permutationen durchprobiert werden. Wegen der Verschachtelung der beiden Wiederholungsoperatoren '*' und '+' sind dies extrem viele. (drei Zeichen sind eine Folge aus einem und zwei Zeichen oder eine Folge aus zwei und einem Zeichen). Eine solches Durchprobieren (Backtracking) führt zu einer sehr langsamen Erkennung des gesamten Ausdrucks. Friedl gibt ein Beispiel, bei dem die Auswertung eines derartigen Ausdrucks die Lebenszeit des Programmierers übersteigen würde.

Das *ewige Matching* kann vermieden werden, wenn der obige Ausdruck folgendermaßen umformuliert wird:

```
"([^"\\\r\n]*(\\.[^"\\\r\n]*)*)"
```

Allgemein unterliegt der neue Ausdruck dem Schema:

*öffnend normal * (speziell normal *)* schließend*

wobei der Anfang von *speziell* und *normal* nicht gleich sein darf.
(*öffnend == schließend == '')*

Hier verbietet *normal* einen Backslash: [^"\\\r\n]
und *speziell* fordert ihn: \\.

Hier gibt es an jedem Punkt der Erkennung nur eine klare Alternative. Tatsächlich ist das Friedl Schema eine LL(1)-Optimierung.

17.10 Interpreter

Ein Interpreter ist ein Programm, dass einen Programmcode parst und dann unmittelbar ausführt. Im Gegensatz zu einem Compiler erzeugt er keinen speicherbaren Maschinencode.

17.11 Lexikalische Analyse

Die lexikalische Analyse zerlegt einen Quelltext in seine elementaren Textbestandteile, d.h. in Lexeme, die als Token interpretiert werden.. Dieser Prozess geht dem eigentlichen Parsen des Textes, bei dem der grammatikalische Zusammenhang der Token analysiert wird, um mindestens einen Schritt voraus.

17.12 LL(k)-Grammatik

Eine Grammatik heißt LL(k) (= von links nach rechts mit linkskanonischen Ableitungen und dem Vorgriff um k Symbole deterministisch erkennbar), wenn man bei der Topdown-Analyse stets anhand der nächsten k Symbole entscheiden kann, wie die Analyse fortzusetzen ist. Insbesondere heißt eine Grammatik LL(1), wenn zu dieser Entscheidung stets das nächste Token ausreicht.

17.13 Numerische Systeme

Wir sind es gewohnt, Zahlen mit 10 verschiedenen Ziffern anzugeben. Computer kennen jedoch nur 0 und 1. Zum Beispiel ist die Dezimalzahl 156 im binären Computerformat ausgedrückt:

10011100

Weil solche Binärzahlen sehr leicht in oktale und hexadezimale Zahlen umgewandelt werden können, gibt es in C++ Konventionen zur Darstellung von Zahlen dieser Systeme. Führende Nullen werden verwendet, um eine oktale Konstante zu bezeichnen, und Zahlen, die mit "0x" beginnen, bezeichnen hexadezimale Konstanten.
Die Umwandlung der obigen Binärzahl in eine Oktalzahl erfolgt durch Auftrennung des binären Werts in 3'er Gruppen und nachfolgender Ersetzung jeder dieser Gruppen gemäß der unten stehenden Tabelle:

10 011 100

2 3 4

In C++ wird der resultierende Wert oktal dann so geschrieben: **0234**

Das Verfahren, die Binärzahl in eine hexadezimale Darstellung umzuwandeln, ist analog, aber Sie müssen dieses Mal die Ziffern in 4'er-Gruppen trennen:

1001 1100

9 C

In C++ wird der resultierende Wert hexadezimal dann so geschrieben: **0x9C**

Escape-Sequenzen sind eine ähnliche Darstellungsweise von Zeichen im Oktal- bzw. Hexadezimal-Format.

Binär	Oktal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Binär	Hexadezimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

17.14 Parser

Ein Parser im engeren Sinne ist ein Programm, das die Syntaxanalyse eines Textes durchführt. Im weiteren Sinne wird das Wort Parser in dieser Hilfe häufig so gebraucht, dass er die lexikalische Analyse und die semantischen Aktionen mit umschließt.

Im TextTransformer spielen mehrere Parser (i.w.S.) eine Rolle. Welcher gemeint ist ergibt sich häufig nur aus dem aktuellen Zusammenhang. Der TextTransformer ist ein Parsergenerator, d.h.

1. der TextTransformer verfügt über einen Parser, der Skripte parst und
2. der TextTransformer erzeugt aus den geparsten Skripte neue Parser. Der erzeugte Parser ist entweder
   a) im Arbeitsspeicher zur Transformierung geladener Texte vorhanden oder
   b) als Parserklasse in Quellcode-Form.

17.15 Parsergenerator

Ein Parsergenerator ist ein Programm, dass aus einer einfachen Grammatikbeschreibung einen Parser erzeugt.

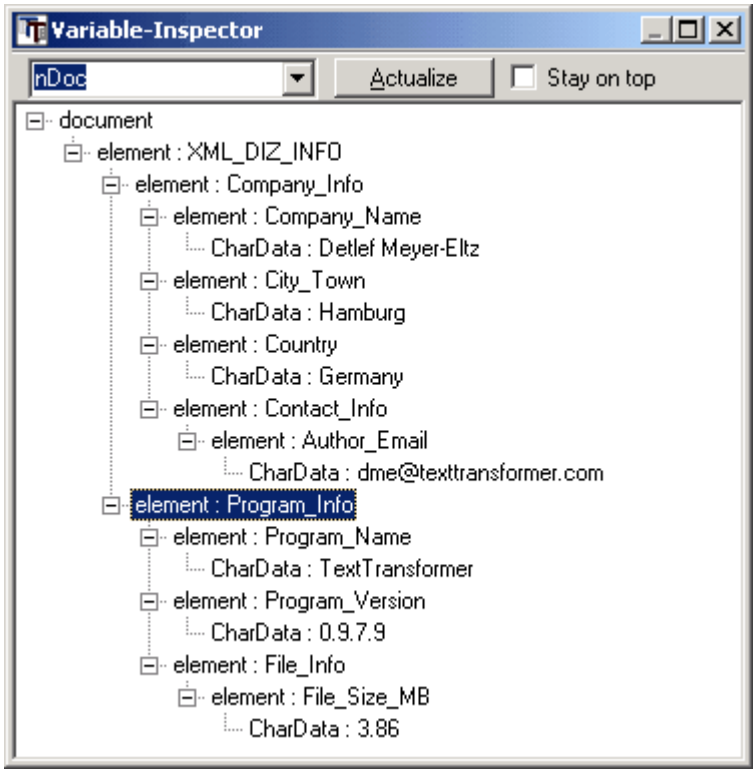
17.16 Parse Trees and AST's

In einer Baumstruktur kann der Eingabetext in einer der Grammatik entsprechenden Struktur abgebildet werden. Ein derartiger Baum wird *Parse tree* oder *AST (abstract syntax tree)* genannt.
Der Vorteil der Erzeugung solcher Baumstrukturen gegenüber der unmittelbaren Übersetzung (*one-pass compiler*) ist:

- Die Daten können mehrmals durchgegangen werden, ohne die Eingabe erneut zu parsen.
- Transformationen können zunächst am Baum selbst durchgeführt werden.
- Auswertungen können in beliebiger Folge durchgeführt werden.

Allgemein besteht ein Baum aus einer Anzahl miteinander verknüpfter Knoten (node).

In der folgenden Abbildung ist eine Baumstruktur zu sehen, wie sie im Variablen-Inspektor angezeigt werden kann.



An Hand dieser Abbildung sollen die Verhältnisse der Knoten zueinander veranschaulicht werden.

Jeder Knoten, der Kind-Knoten besitzt, ist durch ein kleines Quadrat gekennzeichnet, die ***Blätter*** des Baums besitzen kein solches Quadrat. Die **Beschriftung** der Knoten besteht aus ihrem **Label**, gefolgt von einem Doppelpunkt und ihrem **Wert**. In der Abbildung haben alle Knoten bis auf einen das gleiche Label: *element*.

Wurzel- oder ***Root***-Knoten des Baums ist der Knoten an der Spitze des Baums mit dem label: *document*.

Greift man den Knoten mit dem Wert ***Program_Info*** heraus, so steht er in folgenden Beziehungen:

Eltern- oder ***Parent***-Knoten ist: element : XML_DIZ_INFO
Erster Kind- bzw. ***first Child***-Knoten ist : element : Program_Name
Letzter Kind- bzw. ***last Child***-Knoten ist: element : File_Info
Nächster Geschwister- bzw. ***next Sibling***-Knoten gibt es nicht.
Vorheriger Geschwister- bzw. ***previous Sibling***-Knoten ist: element : Company_Info
Erster Geschwister- bzw. ***first Sibling***-Knoten ist: element : Company_Info.
Letzter Geschwister- bzw. ***last Sibling***-Knoten ist: element : Program_Info
Unterster erster Kind- bzw. ***bottom first Child***-Knoten ist: CharData : TextTransformer
Unterster letzter Kind- bzw. ***bottom last Child***-Knoten ist : CharData : 3.86

Vom **Wurzelknoten** aus betrachtet ist
Unterster erster Kind- bzw. ***bottom first Child***-Knoten ist: CharData : Detlef Meyer-Eltz
Unterster letzter Kind- bzw. ***bottom last Child***-Knoten ist : CharData : 3.86

Durchlaufen eines Baums

Ein Baum kann in **ab- oder aufsteigender Richtung** durchlaufen werden. Beginnend mit dem Wurzelknoten werden die Knoten in absteigender Richtung genau in der Reihenfolge besucht, die ihrer Zeilennummer in obiger Abbildung entspricht:

document :
element : XML_DIZ_INFO
element : Company_Info
element : Company_Name
CharData : Detlef Meyer-Eltz
element : City_Town
CharData : Hamburg
element : Country
CharData : Germany
element : Contact_Info
element : Author_Email
CharData : dme@texttransformer.com
element : Program_Info
element : Program_Name
CharData : TextTransformer
element : Program_Version
CharData : 0.9.7.9
element : File_Info
element : File_Size_MB
CharData : 3.86

In aufsteigender Richtung ist die Folge genau umgekehrt.

17.17 Syntax

Die Syntax ist die Lehre vom Bau einer Sprache. Sie sieht von der Bedeutung und dem Kontext sprachlicher Ausdrücke ab und betrachtet lediglich deren strukturelle Beziehungen .

17.18 Startregel

Die Startregel ist die Produktion, mit der die Bearbeitung des Textes beginnen soll. Ausgehend von der Startregel werden die jeweils anstehenden Unterregeln aufgerufen. Die Startregel ist daher allen anderen Produktionen der Transformation übergeordnet.

17.19 Steuerzeichen

Steuerzeichen dienen dazu die Ausgabe der übrigen Zeichen auf dem Bildschirm zu steuern. So sorgt beispielsweise das zu den Steuerzeichen gehörige Zeilenumbruchszeichen dafür, das der nachfolgende Text in einer neuen Zeile dargestellt wird. Zur Bezeichnung der Steuerzeichen bedient man sich jeweils eines Buchstabens des Alphabets, dem der umgekehrten Schrägstrich '\' (der "Backslash") vorangestellt wird.

17.20 Textdateien

Für den Informatiker ist Text schlicht eine Aneinanderreihung von Buchstaben oder allgemeiner gesagt von Zeichen; d.h. in Text ist eine Zeichenkette. Das Reservoir an Buchstaben bzw. Zeichen mit denen ein Computerprogramm umgehen kann ist durchnumeriert. Jedes Zeichen wird so im Speicher eines Computers als eine Nummer festgehalten. In dieser Aufzählung enthalten sind die sogenannten Steuerzeichen. Welcher Buchstabe welche Nummer erhält hängt von dem verwendeten Standard des Betriebssystems oder Landes ab. Der wichtigste Standard, der auch im TextTransformer verwendet wird, definiert den ASCII-Zeichensatz.

In Zeichenfonts sind den Zeichen wiederum Bilder zugeordnet, mit denen sie auf dem Bildschirm angezeigt werden können. Tatsächlich ist nicht in jedem Font jedem Zeichen ein Bild zugeordnet, z.B. wenn es länderspezifisch ist. Kommt ein solches Zeichen im Text vor, so wird es mit einem Platzhaltersymbol dargestellt: einem leeren Quarat.

17.21 Topdown-Analyse

Das Prinzip der (deterministischen) Topdown-Analyse besteht darin, von der allgemeinsten Regel - der Startregel (top) - auszugehen, und zu testen welche ihrer Unterregeln (down) auf den nächsten Textabschnitt passt, usw..
Im Unterschied hierzu geht die Bottomup-Analyse vom Text aus, und sucht nach einer passenden Regel.

17.22 Token und Lexeme

Token sind die elementaren Textbestandteile, in die die lexikalische Analyse einen Text zerlegt. Typische Token sind Worte, Zahlen, Interpunktionszeichen etc.. Token einer Programmiersprache sind z.B. Schlüsselworte wie "double" oder "while". Im Falle der Schlüsselworte sind gibt es eine 1:1-Beziehung zwischen den Token und den **Lexemen**, d.h. den Textabschnitten, die das Token repräsentieren. Im Falle z.B. einer Zahl gibt es zu dem Token viele Lexeme; z.B. "12", "14.8" oder "1001". Derartig allgemeine Token werden daher durch Textmuster beschrieben. Im TextTransformer erfolgt die Musterbeschreibung durch reguläre Ausdrücke.
Ein Problem bilden sich überlappende Token wie z.B. "<" und "<=". Der TextTransformer wählt in einem solchen Konfliktfall automatisch das längere Lexem, also "<=".

Welche Textbestandteile Token sind, hängt letztlich von der Interpretation des Textes ab.

17.23 Unicode

Unicode ist ein Zeichencodierungsstandard, der vom Unicode Consortium entwickelt wurde. Unicode kann mit nur einem Zeichensatz fast alle geschriebenen Sprachen der Welt wiedergeben.

Die einzelnen Zeichen können aber nicht mehr alle durch ein einzelnes Byte dargestellt werden, wie es beim ASCII- bzw. ANSI-Code der Fall ist. Es gibt verschiedene Methoden, wie die Zeichen mittels mehrerer Bytes repräsentiert werden können.

- Zur Darstellung jedes einzelnen Zeichens werden zwei (oder vier) Bytes verwendet. Diese Methode wird im Windows-Betriebssystem angewandt.
- Verschiedene Zeichen werden je nach ihrer allgemeinen Bedeutung mit einer unterschiedlicher Anzahl an Bytes codiert. Ein sehr verbreiteter Standard, der diese Methode anwendet ist UTF-8. Im UTF-8 codierten Unicode belegen die ersten 128 Zeichen des ASCII-Codes jeweils nur ein Byte. ASCII-Code und Unicode sind hier identisch. Die im ANSI-Code nachfolgenden 128 Zeichen werden in UTF-8 durch jeweils zwei Bytes repräsentiert und alle weiteren Zeichen benötigen für ihre Darstellung noch mehr Bytes.

Beispiel:

Wird eine UTF-8 codierte Datei im ANSI-Modus geöffnet, so erscheint das Wort "für" als:

fÃ¼r

Der Umlaut 'ü' belegt in UTF-8 zwei Bytes, die im ANSI-Modus als zwei Zeichen dargestellt werden. Öffnet man hingegen die Datei im UTF-8-Modus, wird das Wort korrekt angezeigt.

17.24 Zeilenumbrüche

Zeilenumbrücke werden in Textdateien durch besondere Steuerzeichen dargestellt. In den Windows Betriebssystemen ist dies eine Kombination aus dem Carriage-Return-Zeichen '\r' und dem Linefeed-Zeichen '\n'. Werden z.B. die einzelnen Zeichen des dreizeiligen Texts:

1. Zeile
2. Zeile
3. Zeile

In einer Tabelle aufgelistet, sieht diese folgendermaßen aus:

1	.	Z	e	i	l	e	\r	\n
2	.	Z	e	i	l	e	\r	\n
3	.	Z	e	i	l	e		

In Unix-Betriebssystemen ist es üblich statt der '\r\n'-Kombination allein das Linfeed-Zeichen zu verwenden. Der gleiche Text ist dort also etwas kürzer:

1	.	Z	e	i	l	e	\n
2	.	Z	e	i	l	e	\n
3	.	Z	e	i	l	e	

Im TextTransformer Editor wird stets die Windows Konvention verwendet. Werden Unix-Texte geladen, so wird automatisch allen Linfeed-Zeichen ein Carriage-Return-Zeichen vorangestellt. Ab Version 0.9.8.1 erfolgt dann ein entsprechender Warnhinweis.

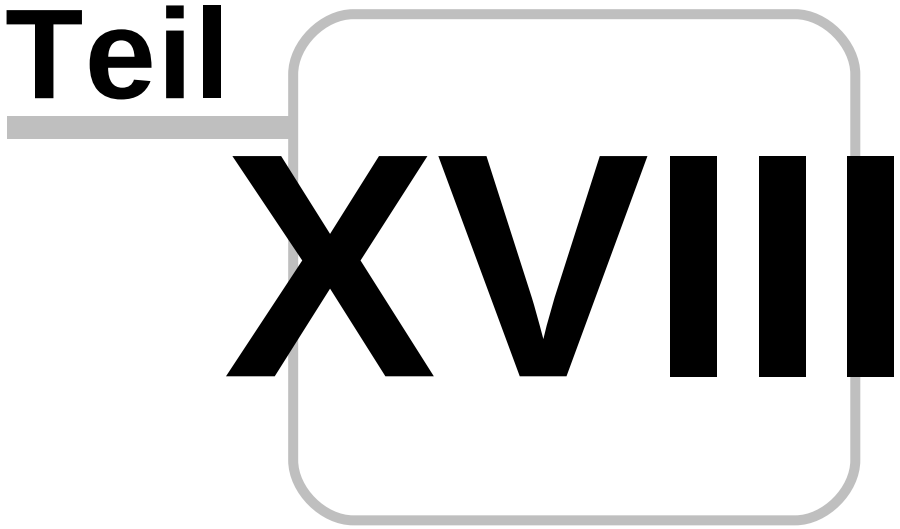
In den meisten Transformations-Projekten werden die Zeilenumbrüche ignoriert. Sind sie aber konstitutiv für die Struktur des zu analysierenden Textes, muss auf den eventuellen Einschub der '\r'-Zeichen geachtet werden. Im Extremfall kann dann die Ausführung eines Projekts verschiedene

Ergebnisse haben, je nachdem, ob sie in der TETRA-Arbeitsoberfäche passiert oder im Transformationsmanager bzw. mit dem Kommandozeilen-Tool, die jeweils die unveränderten Texte behandeln.

Sowohl Windows- als auch Unix-Zeilenumbrüche werden durch folgendes Token erkannt (wenn die Zeilenumbrüche nicht ignoriert werden):

```
EOL  ::= \r?\n
```

TextTransformer



18 Namens-Konventionen

In der Hilfe und in den Beispielen werden nahezu durchgängig einige Namens-Konventionen verwendet, die das Lesen der Skripte vereinfachen sollen. Die Verwendung solcher Konventionen im eigenen Quelltext, kann auch von großem Nutzen sein, wenn man ihn später einmal mit dem TextTransformer bearbeiten will. Die Verwendung der Konventionen steht Ihnen aber völlig frei. Das Funktionieren des TextTransformers hängt davon in keiner Weise ab.

Ausnahme: Der Funktionstabellen-Assistent arbeitet nicht korrekt mit Funktionstabellen zusammen, deren Name nicht mit "m_ft" beginnt.

Variablen wird eine abgekürzte Typ-Bezeichnung vorangestellt.

Typ	Bezeichner	Beispiel
bool	b	bVar
char	c	cVar
int	i	iVar
unsigned int	ui	uiVar
double	d	dVar
str	s	sVar
node	n	nLeaf
vector	v	vParams
map	m	mReplace
cursor	cr	cr1
Funktionstabelle	ft	ftPrint

Den Namen von **Klassen-Variablen** wird ***m***_ vorangestellt.

Beispiel: m_sColor eine Klassen-Variable vom Typ str

Den Namen von **Parameter-Variablen** wird **x** vorangestellt.

Beispiel: xsColor ein Parameter vom Typ str

Index

- B -

- C -

- D -

- E -

- F -

- G -

- H -

- I -

- J -

- K -

- L -

- M -

- N -

- O -

- P -

- Q -

- R -

- S -

- T -

- U -

- V -

- W -

- X -

- Y -

- Z -