TextTransformer 1.7.5

Table of Contents

Part I About this help 16

Part II Registration 18

Part III Most essential operation elements 21

Part IV Most essential syntax elements 23

Part V How to begin with a new project? 25

1 Practice 27

Part VI Introduction 30

1 How does the TextTransformer work? 30
2 Analysis 30
3 Synthesis 31
4 Regular expressions 31
5 Syntax tree 32
6 Productions or non-terminal symbols 33
7 Productions as functions 33
8 Four uses of productions 35
9 Looking-ahead 35
10 Inclusions / comments 37
11 Sub-parser 37
12 Family concept 37
13 Tests 38

Part VII Examples 40

1 Exchange of words 41
Execution of a project 41
Production 43
Analysis step by step 44
Using tokens 45
2 Conversion of an Atari text 47
Tokens 47
Productions 49
Actions 50
Conversion into RTF 51
3 Calculator 54

Tokens 54
Production: Calculator1 55
Production: Expression 56
Productions: Term and Factor 57
Production: Number 59
Return values 59
4 Text statistics 61
Class members 61
Token 62
Productions 63
5 GrepUrls 64
Productions 64
Member variables and methods 65
Put together 66
Search in whole directory 67
6 BinaryCheck 70
Look-ahead 70
Use as preprocessor 71
7 E-mail address 72
Syntax specification 73
Productions and token 74
Detecting a conflict 75
Solving the conflict 76
8 Guard 78
Startrule: guard 79
Copying source text 79
Tokens 80
Productions: block, outer_block 82
Improvement: '{' and '}' in strings 83
9 Bill 84
Production 84
Tokens 85
10 XML 86
ISO-XML 86
XML document 88
Tree generation 90
Tree evaluation 90
Character references 92
Comments and processing instructions 93
Insert client data 94
11 Unit_dependence 96
Productions 96
Containers and parameters 97
Include files 98
12 Java 98
Coco/R adaptation 99
Simple look-ahead production 99
Negative look ahead 99
Complex look ahead 100
Debugging a look-ahead 100
Parse-Tree 102

Function-Table 105
13 C-typedef 108
Typedef 108
Scopes 109
14 TETRA productions 110
15 TETRA-EditProds 110
16 TETRA interpreter 110
17 TETRA import 111
18 TETRA-Management 112
19 Cocor import 113
Ignorable characters 113
Tokens 114
Productions 115
Post processing 115
Semantic actions 116
Part VIII How to ... 118
1 Load data 118
2 Structure data 119
3 Write into additional target files 119
Part IX User interface 121
1 Tool bar 121
2 Main menu 122
Menu: File 122
Menu: Edit 125
Menu: Search 126
Menu: Project 127
Menu: Start 128
Menu: Code generation 129
Menu: Options 130
User data 130
Options of the user interface 131
Transformation 132
Editing 133
View 133
Layouts 134
Environment options 135
CONFIG 135
EXTENSIONS 136
FRAMES 136
PATH 136
File filter 137
Project options 137
Names and Directories 138
Start rule 138
Test file 138
Preprocessor 138
Frame path 139

Parser/Scanner ... 139
Ignorable characters ... 139
Case sensitive ... 141
Word bounds ... 141
Parameter and {{...}} ... 142
Global scanner ... 142
Look-ahead buffer ... 144
Start parameters ... 144
Inclusions (comments) ... 145
Encoding ... 145
xerces DOM ... 146
DTD ... 149
Warnings/Errors ... 150
Stack maximum ... 150
Code generation ... 150
const ... 151
Use wide characters ... 151
Only copy all code ... 152
Characters and increment of indentation ... 152
Operating system ... 152
Plugin type ... 152
Template parameter for plugin character type ... 153
Version information ... 153
Local options ... 154
Local options ... 154
Menu: Windows ... 155
Docking Windows ... 156
Caption Dialog ... 158
Window list ... 159
Customize layout ... 160
Save Layout ... 161
Restore default layout ... 162
Menu: Help ... 162
Feedback ... 163
Wizards ... 163
New project wizard ... 163
Multiple replacements of words ... 164
Multiple replacement of strings ... 165
Multiple replacements of characters ... 165
CSV-wizard ... 167
Creating a line parser from an example text ... 168
Header/Chapters/Footer ... 169
Actions ... 170
Creating a production from an example text ... 171
Parameter-Wizard ... 172
Tree-Wizard ... 173
Tree type ... 173
Function-Table-Wizard ... 174
Quick wizard for function tables ... 176
Input tables ... 177
Regex test ... 178
Character class calculator ... 180
ANSI table ... 184
3 Script management and parsing ... 184

Tool bar and menu **186**
Insert **187**
Delete **188**
Edit **188**
Cancel **188**
Accept **188**
Rename **189**
Navigation **189**
Parse/Test single script **189**
Parse/Test all connected scripts **189**
Parse/Test all scripts **190**
Error messages **190**
Clear semantic code **191**
Import **192**
Export **194**
Collapsing semantic code **194**
4 Debugging and executing **195**
Source text **196**
Section of text **197**
Enabling actions **197**
Choosing a start rule **198**
Interactive change of a start rule **198**
Change of the start rule **199**
Parse start rule **199**
Syntax tree **200**
Pop up menu 202
Show first sets 203
Start mode **208**
Execution step by step **209**
Execute a look-ahead step-by-step **210**
Execution at a stretch **211**
Checking success **211**
Reset **212**
Mark recognized/expected token **212**
Breakpoints **213**
Text breakpoint 213
Node breakpoint 214
Recognized token **215**
Stack window **215**
Variable-Inspector **216**
To the actual position **218**
Info box **219**
Log window **219**
5 Transformation of groups of files **219**
Transformation manager **219**
Defining a new filter 220
Selecting source files 220
Transformation options 222
N:N Transformation 223
Select target directory 223
Setting pattern for the target files 226
Backup 226
N:1: Transformation 227
Preview of the target files 228

Start the transformation ... 229
Results ... 229
Report ... 230
Corrections ... 230
Roll back ... 231
Management ... 231
Command line tool ... 232
Parameter ... 232
6 Keyboard shortcuts ... 234
Block commands ... 235
Part X Scripts 238
1 Token definitions ... 238
Input mask for a token ... 238
Name ... 239
Return type ... 239
Parameter declaration ... 239
Comment ... 239
Text ... 240
Semantic action ... 240
Literals ... 241
Named literals ... 241
Regular expressions ... 242
Single characters ... 243
Meta-characters ... 243
Special characters ... 243
Sets of characters ... 244
Character classes ... 244
Locale dependant features ... 245
Collating elements ... 246
Equivalence classes ... 246
Collating Element Names ... 246
Wildcard ... 248
Anchors ... 248
Concatenation ... 250
Groupings ... 250
Alternatives ... 250
Repeats ... 251
Macros ... 252
boost regular expression library ... 253
Predefined tokens ... 253
Identifier ... 254
Words ... 255
Numbers ... 256
Quotes ... 257
Dates ... 257
Comments ... 258
Ignorable ... 259
Line break ... 259
Binary null ... 260
Addresses ... 260
Data field ... 260
Placeholder ... 261

2 Productions 262
Input mask for a production 262
Name 262
Return type 263
Parameter declaration 263
Comment 264
Text 264
Elements 265
Concatenation 266
Alternatives 266
Grouping 267
Repeats 268
BREAK 269
EXIT 270
EOF 271
ANY 271
Options 272
SKIP 273
Options 275
IF...ELSE...END 277
WHILE...END 279
Actions 280
Transitional action 281
Calling parameters 282
3 Class elements and c++ instructions 282
Input mask for class elements 283
Name 283
Type 284
Parameter 284
Comment 285
Text/Initialization 285
List of all instructions 286
Interpreted C++ instructions 292
C++ 292
Variable types 293
bool 293
char 294
int 294
unsigned int 294
double 294
str 295
Searching 296
Container 298
vector 299
Stack 301
map 303
cursor 306
General cursor methods 307
Function table 309
node / dnode 311
node: Construction 312
node: Information 313
node::npos 316
node: Neighbors 316

node: Searching ... 318
node: Sorting ... 320
dnode specials ... 320
const ... 321
Operators ... 321
Arithmetic operators ... 321
Assignment operators ... 322
Relational operators ... 323
Equality operators ... 323
Logical operators ... 324
Bitwise operators ... 324
Conditional operator ... 325
Control structures ... 325
if, else ... 325
for ... 326
while ... 326
do ... 327
switch ... 327
Output ... 327
out ... 328
log ... 329
Binary output ... 329
return ... 330
break ... 330
continue ... 330
throw ... 330
String manipulation ... 331
stod ... 331
stoi ... 332
hstoi ... 332
stoc ... 332
dtos ... 333
itos ... 333
itohs ... 334
ctohs ... 334
ctos ... 334
to_upper_copy ... 335
to_lower_copy ... 335
trim_left_copy ... 335
trim_right_copy ... 336
trim_copy ... 336
File handling ... 337
basename ... 337
extension ... 338
change_extension ... 338
append_path ... 339
current_path ... 339
exists ... 340
is_directory ... 340
file_size ... 340
find_file ... 341
load_file ... 341
path_separator ... 342
Formatting instructions ... 342

How it works ... 343
Examples ... 343
Syntax ... 344
Methods ... 346
Other functions ... 346
clock_sec ... 347
time_stamp ... 347
random ... 348
Parser class methods ... 349
Parser state ... 349
Sub-expressions ... 354
Plugin methods ... 355
Source and target ... 356
Start parameters ... 357
Redirection ... 358
xerces DOM ... 358
Indentation stack ... 359
Text-scope stack ... 361
Dynamic scanner ... 362
Error handling ... 364
Calling a production ... 365
Sub parser ... 365
Look-ahead ... 366
Events ... 368
4 Test scripts ... 369
Name ... 369
Group ... 370
Comment ... 370
Input ... 370
Code ... 370
Expected output ... 370
Test output ... 371
Error expected ... 371
Part XI Algorithms 373
1 Scanner algorithm ... 373
2 Parser algorithm ... 374
3 Token sets ... 375
Part XII Grammar tests 379
1 Completeness ... 379
2 Reachable rules ... 379
3 Derivable rules ... 380
4 Non-circularity ... 380
5 LL(1)-Test ... 380
6 Warnings ... 381
7 Nullability ... 381
8 Start of several alternatives ... 382
9 Start and successor of nullable structures ... 382

10 SKIP node with SKIP neighbors ... 384
11 Different SKIP followers ... 384
12 Different ANY followers ... 384
13 Left recursion ... 384
14 Circular look-ahead ... 385

Part XIII Code generation 387

1 Code frames ... 387
Name of the parser class ... 387
Header frame ... 388
Implementation frame ... 390
main-file frame ... 393
Project specific frame ... 396
jamfile ... 397
2 Supporting code ... 397
Code directory ... 398
CTT_Parser ... 398
Methods ... 399
CTT_ParseState ... 402
CTT_Scanner ... 403
CTT_Tst, CTT_TstNode ... 403
CTT_Match ... 403
CTT_Token ... 403
CTT_Buffer ... 404
CTT_Guard ... 404
CTT_Mstrstr ... 405
CTT_Mstrfun ... 406
CTT_Node ... 406
CTT_DomNode ... 406
CTT_ParseStatePluginAbs ... 407
CTT_ParseStatePlugin ... 407
CTT_ParseStateDomPluginAbs ... 407
CTT_ParseStateDomPlugin ... 407
CTT_RedirectOutput ... 407
CTT_Indent ... 407
CTT_Xerces ... 408
3 Error handling ... 408
4 Compiler compatibility ... 410
5 License ... 410

Part XIV TetraComponents 413

Part XV Messages 415

1 Unknown symbol: "xxx" ... 415
2 "X": can't derive to terminals ... 415
3 Circular derivation: "X" . "Y" ... 415
4 "X" is nullable ... 415
5 LL(1) Error: "X" is the start of several alternatives ... 415

6 LL(1) Warning: "X" is the start and successor of nullable structures 415
7 "X" is a SKIP node with SKIP neighbors 415
8 Nullable structure in a repetition or option 416
9 "X" is used circulary in a look-ahead 416
10 Inclusion not found 416
11 Conflict with an inclusion 416
12 No matching next token found 416
13 The rest of the text consists of ignored chars 416
14 SKIP token matches at actual position 417
15 "SKIP ANY" is not possible 417
16 Matching but not accepted token 417
17 Matching token not in first set 418
18 Matching look-ahead xxx cannot start with yyy 418
19 Unexpected symbol in 418
20 Parenthesis are needed 418
21 Unexpected method (also might be 419
22 "X" expected 419
23 Incomplete parse 420
24 Missing closing quotation mark 420
25 Literal tokens may not be empty 420
26 Continuation with c++ code expected 420
27 The type of the function xxx doesn't match the function table 420
28 No default function is defined for function-table 421
29 In a const parser you have to call the according method of State 421
30 Sub-expressions (> 0) are not stored in the la-buffer 421
31 A production cannot be used as an inclusion 421
32 Inclusion with paramters 421
33 Inclusions don't work with a la-buffer 422
34 State parameter is required 422
35 Empty alternative 422
36 Error while parsing parameters 422
37 Mismatch between declaration and use of parameters 422
38 Wrong number of (interpretable) arguments 423
39 Not const method 423
40 Maximum stack size of "x"exceeded 424
41 Error on parsing parameters of the parser call 424
42 There is at least one path on which no string value is returned 425
43 Recognized, but not accepted token 425
44 BREAK outside of a loop 426

45 Standstill ... 426
46 Standstill in look ahead ... 426
47 Unknown identifier : xxx ... 427
48 It's not possible to convert "xxx" to "yyy" ... 429
49 No return type declared ... 429
50 "X" cannot be applied on "Y" ... 429
51 break or continue instruction at invalid position ... 429
52 forbidden transitional action ... 430
53 Error output programmed from the user ... 430
54 Cannot add branch ... 430
55 Token error ... 430
56 Matches empty string ... 430
57 Token is defined as string and as token with an action ... 430
58 boost::regex error ... 431
59 System overlap ... 431
60 Token action or member function cannot be exported ... 431
61 Only code for initializations is allowd here! ... 431
62 Parameters and local variables may not be used in a look-ahead production! 431
63 Encoding cannot be written into the output window of the IDE ... 432
64 An invalid or illegal XML character is specified ... 432
65 TextTransformer not registered ... 432
66 Internal error: 433
67 No help ... 433

Part XVI References 435

1 References ... 435

Part XVII Glossary 440

1 First set ... 440
2 ASCII/ANSI-set ... 441
3 Backtracking ... 441
4 Binary file ... 442
5 Compiler ... 442
6 Control characters ... 442
7 Debug ... 442
8 Deterministic ... 442
9 Escape sequences ... 442
10 Friedl scheme ... 444
11 Interpreter ... 444
12 Lexical analysis ... 444

13 LL(k)-grammar 445
14 Numeric systems 445
15 Parser 446
16 Parser generator 446
17 Parse Trees and AST's 446
18 Syntax 448
19 Start rule 448
20 Text file 449
21 Top down analysis 449
22 Token and lexemes 449
23 Unicode 449
24 Line breaks 450

Part XVIII Naming conventions 453

Index 454

Part I

1 About this help

TextTransformer is made for many kinds of apllication and users. This help was therefore written so that it should be comprehensible also of programming newcomers.

The examples are constructed like a tutorial and demonstrate the possibilities of the TextTransformer. It is suggested that you experiment with these before developing your own applications.

The introduction presents a short view on the working method of TextTransformer and explains some basic notions.

A wizard helps to create new projects. It is suitable to play with the options of this wizard, to get a first impression of the TextTransformer.

In this help the expression **TETRA** will be used sometimes as abbreviation of the TextTransformer program.

TETRA is a program to create, test and execute rules and instructions, which transform a source text into a target text.

The TextTransformer exists in three versions. In contrast to the free version the standard version allows index operations, as for example, the access of sub-expressions of regular expressions and the use of container classes. Furthermore the transfomation manager, the variable-inspector and most wizards are provided only for the standard and professional version. Only the professional version can generate C++ source code.

In **blue font** the sections of this help are marked, which only concern the professional version.

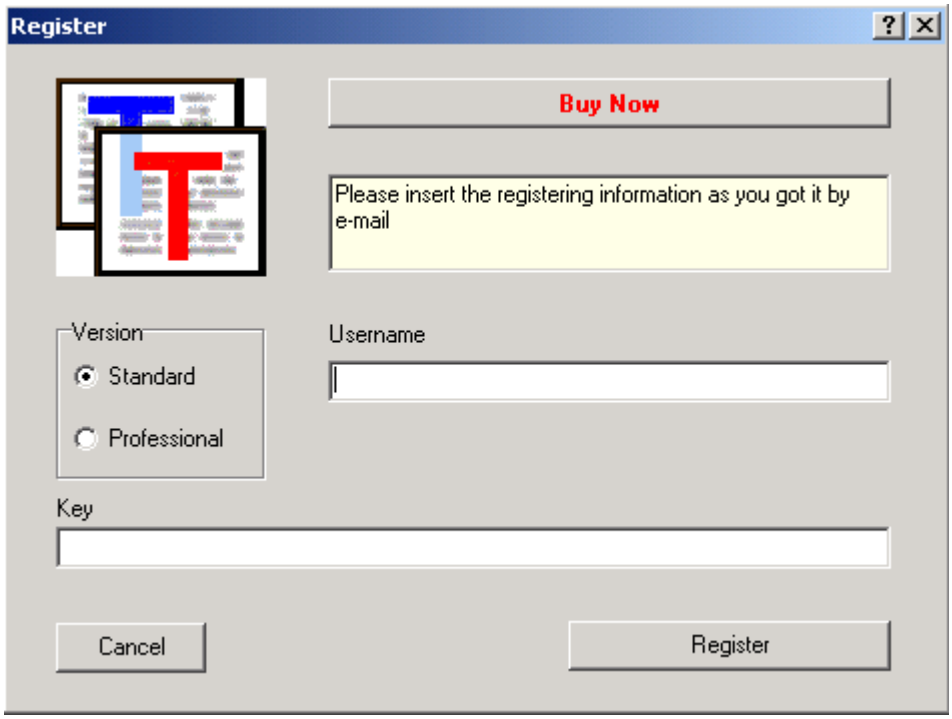
The **red font** is used to emphasize warnings or other important remarks.

Part II

2 Registration

The TextTransformer is sold exclusively by Internet. There is no CD and no separate manual.

The **registration** of the TextTransformer, i.e. the activation of the features of the standard or professional version, can be done by the menu: Help->*Registration*.



For the registration of the **Standard version** you must transmit a **user name** (at least eight characters) and your address details and the details on the method of payment. For the registration of the **Professional Version** in addition a **program ID** (see below) is required.

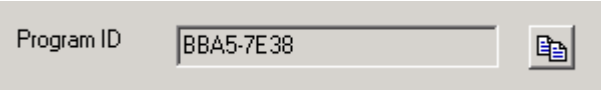
Forms for the corresponding inputs are displayed in your Internet browser, if you are on line and click the **Buy Now** button.

After the check of your credit card has been carried out, an **e-mail** which includes the registration data (user name and key) is sent to you automatically. These must be assigned to the appropriate fields of the dialog box shown above. But first select, whether a registration of the **standard or the professional version** shall be carried out. **User name** and the **key** then have to be copied unchanged from the e-mail into the corresponding entry fields of the dialog box.
Then the **button *Register*** will close the dialog automatically and a message appears, which confirms the success of the registration.

Program ID for the registration of the Professional version

The program ID that is required for the registration of the Professional version is shown as soon as you select the button Professional in the dialog box. An additional field appears with a combination

of numbers and letters.



This program ID is copied into the clipboard if you click the button at the right.
The program ID is specific for your hardware configuration. The registered professional version can be executed only on the computer on which it originally was installed.

It is important to know that if you are downloading the TextTransformer to use the Professional version on a different computer than the one on which you originally downloaded it, you should transfer it immediately onto removable media, and *not* register it on the first computer.
(For the standard version there is no such restriction. You can arbitrarily transfer it.)

Upgrade to Professional Version

If you have registered the Standard Version of the TextTransformers, in the dialog appears a button, by which you can upgrade your license to the Professional Version, instead of the **Buy Now** button

Part III

3 Most essential operation elements

It is suggested that you read the introduction and experiment with the wizard for new projects and with the examples, before you begin to develop your own projects.

1.
Either open an existing project and an input text by the file menu or write a text directly into the input window and on the production page

 create a new production and
 confirm.

2.
A program only can be parsed and executed, if in the selection box of the tool bar

a start rule is set and only, if the actions are activated, there will be an output.

3.
To parse a production, use the button:

This button exists as well on the tool bar, where it will parse the start rule of the project as on the production page, where it will treat the actual production as start rule.

4.
Execution of a TextTransformer program is triggered either by:

 in the slow debug mode

with the possibility of break points and complete error messages, or by:

 fast in the execution mode.

5.
Return from the debug or the execution modus into the edit mode by

 Reset.

Part IV

4 Most essential syntax elements

Parser and Scanner

Grouping	(...)
Alternative	|
Option	?
Optional repeat	*
Repeat	+

Arguments are passed to productions or tokens inside of brackets '[' and ']', which follow the name of the production of token; e.g. Name[iCount]

Interpreter

The syntax of the interpreter is simple c++ syntax.

Tip: Write simple code and better use two instructions than one.

C++ code is inserted into the parser description inside of special brackets:

only executable in the interpreter	{- ... -}
only for the c++ export	{_ ... _}
for interpreter and export	{= ... =}
according to project options	{{ ... }}

Frequently used instructions or expressions are:

Writing into the output	out << value;
last recognized text	State.str()
ignored characters before	State.str(-1)
last recognized text, ignored characters before inclusive	State.copy()

Part V

5 How to begin with a new project?

It is suggested that you read the introduction and experiment with the wizard for new projects and with the examples, before you begin to develop your own projects.

For many programming languages and formats one can find ready grammars in the Internet. If such a description exists, you often can translate it into the syntax of the TextTransformer quite easily. A half automatic translator for Coco/R grammars belongs to the examples of the TextTransformer package. In the examples: E-mail address and XML, is demonstrated, how available syntax specifications can be converted into TextTransformer programs.
If no syntax description exists, you have to create it your own. There are some rules and experiences which can serve as a guide at the construction of a new project.

1. Set the required project options!

E.g. it is very important already at the beginning of the development of a new project, to select the characters, which don't have a meaning for parsing the texts. Per default the line feed and the line break characters are amongst them. This setting must be changed, if line breaks have to be recognized.
Another important decision is, whether all literal tokens should be tested or not. Rule of thumb: All literals should be tested for formalized languages with defined key words at significant positions. Only the expected literals should be tested otherwise The local options also can if necessary be adapted respectively.

2. At first design the parser without semantic actions!

For the construction of the parser it often will be necessary or appropriate to rearrange productions and to simplify complex productions by definition of sub-productions. If the parser already contained a semantic code, this had to be adapted newly at each of these changes.

3. Develop top down!

Start with the most general production, the start rule that shall recognize the complete text, and then take the start rule to pieces of sub-productions which shall recognize principal parts of the text. According to the same principle the sub-productions then further can be refined. If e.g. a book shall be parsed, then the start rule would be:

```
Book ::= SKIP  // recognizes the whole text
```

After the first improvement:

```
Book ::= SKIP? Chapter+

Chapter ::= TITLE SKIP
```

TITLE here stands for a regular expression, which unmistakably distinguishes a chapter heading from other text components.
Remark: Such an expression doesn't exist certainly for all books. The book is used as an example

of a text structure, which everybody knows. The book parser works only for syntactically ideal books. (e.g. TITLE ::= \d\.[^\r\n]+ // if you take the text of this page as a "book".)

The Chapter production can further be refined now:

```
Chapter ::= TITLE Paragraph+

Paragraph ::= EOL+ SKIP

EOL ::= \r?\n  // end of line
```

The advantage of this top down procedure is, that in every stage of the development the current parser can be tested at all "books". Possible faults can so already be discovered in an early stage of development.

Note: With the transformation manager many examples can be tested as a batch. If such a test fails, the corresponding text can be opened with a click in the IDE.

4. Choose the kind of transformation!

There in principle are three ways how the parser can be completed to a transformation program. They differ in what is done with the recognized text sections.

a) text sections are immediately processed and written into the output.

b) text sections are, written into variables and these are returned or passed as (reference-) parameters to other productions, where they can be evaluated or combined to new values.

c) a parse tree is produced and the processing of the text sections are carried out after the complete text was parsed.

The last method is the most variable since all text sections still can in principle be accessed and since with the parsing tree a different output can be caused, depending on the used function table. If a translator shall be developed, which shall convert one format into several output formats, then the use of a parse tree is nearly indispensable. The development of such a translator is, however, much more difficult than the direct processing of the source text with one of the two other methods.

If the order in which the processed text sections shall be put out is approximately identically with the sequence in which they were recognized, the first method of direct output is recommend. If recognized text parts must be rearranged or the processing of a part depends from a text that is found later, the second method is recommend.

If you have decided about the way of the transformation, different wizards can help you to insert parameters, variable declarations or tree nodes into the productions.

5. Make a copy program before writing the definite transformation code!

This rule only applies to projects at which the source text shall be modified in some significant places. If at first a program is made, which simply copies the source text, by comparison with the target text can be found easily, whether the output is complete.

How to begin practically

5.1 Practice

1. Create a new project

In the file menu choose the item new project. At first the wizard for new projects appears.

2. Enter names

A project name and a name have to be entered on the first page of the wizard for the start rule. In most cases you should call both same: Book. If this name is entered for the project name to the field, then it appears also at once in the field for the start rule and the next step is shown in the left menu of the wizard: *Project type*.

3. Choose project type

You can go to the page for the project type either by selecting the button Next or by clicking on the menu item. There are four different choices for project types on this page. The last one: "**New project from scratch**" shall be chosen here. As soon as it is selected, the page is changed to the next one *Finish* automatically, where the chosen names are shown again.

4. Save raw project

As soon as the *Finish* button is pressed, a file selection dialog appears where the folder is selected in whom the project shall be saved.

5. Edit start rule: *Book*

When the project was saved, the wizard is closed automatically and you are on the main page of the Tetra IDE. The chosen name *Book* is already shown on the selection box for the start rule in the toolbar. It as well as it is already registered in the project options and it can be seen also in the syntax tree on the right side of the IDE. If it is selected there, then the tab page *Productions* opens and you see the definition of the book production: SKIP. With SKIP every text is recognized. So the definition must be edited.

As suggested - in the theoretical part under item 3 - the definition text of the *Book* production is replaced now by:

```
SKIP? Chapter+
```

At first the expression *Chapter* is represented in a normal font. If the *Chapter* production were already defined, it would be shown in a brown boldface printing. This definition shall be made now.

6. Insert production: *Chapter*

At first a new rule is created with the plus button. Write for its the name *Chapter*, please. The definition text shall be

```
TITLE SKIP
```

After confirmation, the expression *TITLE* is represented in a normal font this time. If you go back to the *Book* production with the back button, you will see that *Chapter* is highlighted there meanwhile. Now *TITLE* has to be defined. The capitalization of the word shall express that it isn't a production but a token. However, such a capitalization isn't necessary.

7. Insert token: TITLE

Tokens have to be defined on the token tab page. The operation is analogous as on the production page here. With the plus button a new token is produced which gets the name *TITLE*. A title might be defined by

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

This expression can recognize headings, which begin with a digit and a dot, followed by an arbitrary sequence of characters up to the line ending. This isn't for certain a general syntactic definition for titles; it only serves as an example.

8. Compile project

The project is complete now and can be compiled. Click on the button *Parse start rule* in the main tool bar.. You can see the structures of the productions in the syntax tree now, too.

9. Save again

Please don't forget to save the project. You can refine the project now as suggested in the theoretical part.

Part VI

6 Introduction

In this help the acronym **TETRA** will be used sometimes as abbreviation of the TextTransformer program. TETRA transforms text; it translates a source text into a target text.

However, what this means, can be explained best by examples.
The transformation can be simple **replacements of words**, for example the replacement of "TextTransformer" by "TETRA". In contrast to a normal text processing, by TETRA not only single pairs of words but also whole lists of such pairs can be processed at a stretch.
A little bit more advanced are **rearrangements of words**. So mailing lists, consisting of lines with name, address and phone number could be transformed to: phone number, name and address.
Another example is the **extraction of data** from a text. For example names and prices of certain products could be extracted from a catalog and arranged to a list.
Also it is possible, to apply certain actions on the extracted data, for example a **calculation**. Prices could be extracted from a bill and summed. In this case the transformation of text would be to build a single sum from a bill. At this example you can see, that the expression "transformation of text" should be understood in a very broad sense.

The previous examples were of quite easy nature. With some practice such transformation programs can be **written in minutes**. Who isn't practiced yet so much, will need some more time, but the time will pass very quickly, because the development of a TETRA program is a **playful pleasure**: eventual errors are signalized immediately and you can try everything just easy, piece by piece.

Its whole strength the TextTransformer shows, when dealing with complex grammars, as for example the **translation of a programming language into a different**.
Finally, TETRA can be used in a totally different way. Instead of analyzing texts with a predefined structure, you can develop just such a structure. By means of TETRA **new programming languages** can be created. For example the central parts of the TextTransformer are made by TETRA itself.

6.1 How does the TextTransformer work?

Two main tasks have to be done, when a transformation program is written:

1. The analysis of the source text
2. The synthesis of the target text

A source text has to be parsed according to its syntactical structure and at the same time it will be recreated as target text in a new form. The user has to formulate as well the syntactical rules of the source as the instructions for the synthesis of the target text.

6.2 Analysis

The analysis of the source text is done in two steps.

The **lexical analysis** takes the source text to words, punctuation marks etc. More general: the

lexical analysis is the recognition of the so-called **tokens**. Tokens, also called **terminal symbols**, consist of one or several characters. Such a sequence can be considered as a pattern of characters, which can be described generally by so-called **regular expressions**.
Depending on the kind of text, these tokens can denote different things. In a mathematical text names, numbers and operators will be considered, in texts of the natural language words, groups of words, sentences or parts of words are basically elements and in records the different fields.
The lexical analysis also will remove meaningless characters from the text, as spaces, tabs comments etc.

The **syntactical analysis** evaluates in which order the token appear in the text.
This order is defined by sequences of alternatives of tokens, which repeatedly follow in the text one after the other. For example, a text simply can be considered as a sequence of lines or as repeated occurrences of groups of words separated by punctuation marks. Also the text can obey a grammar, described by complex rules.
A syntactical rule is named a production or a **non-terminal symbol**.

6.3 Synthesis

The analysis of the text took it to its pieces: the tokens and their sequences.
The synthesis performs some **semantic actions** on the pieces and combines them to a new text.
The instructions for the semantic actions are embedded into the definitions of the productions.

The instructions for the synthesis stem from the programming language c++. A sub-system of c++ is integrated in TETRA as an interpreted language that means a certain set of c++ instructions can be executed directly inside of the TETRA environment. (In contrast to this c++ normally is a language, which must be compiled to a separated executable.) For example, by the integrated instructions, you can chain parts of texts into a new order or replace them by other texts and much more.

In the professional version of TETRA program code can be generated, which can be linked in external applications and can use the possibilities of c++ without limitation.

6.4 Regular expressions

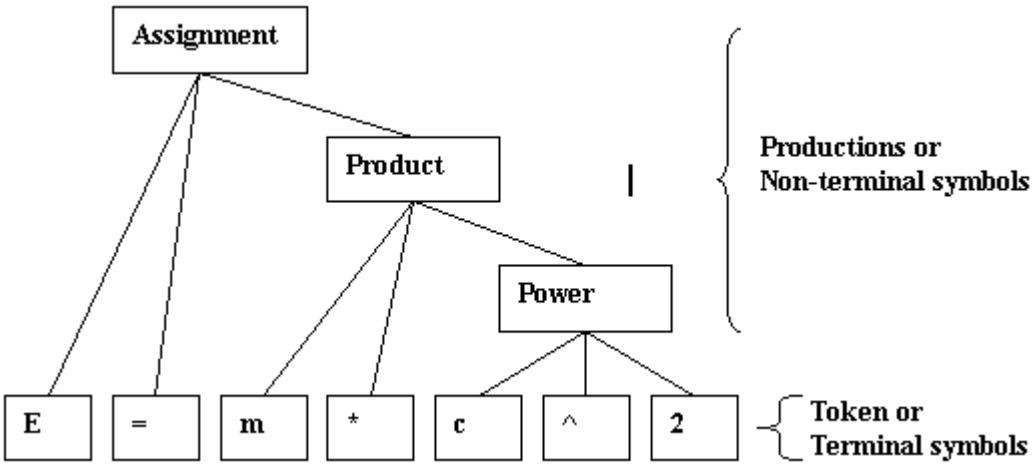
As already mentioned, the pieces to which TETRA takes the text are called *token* or *terminal symbols.* They are identified by *regular expressions.* Regular expressions are well known from many script languages or text processing programs. They extend the possibilities of finding words, because by them not only special, single words can be found, but also words contained in a group of words of a common pattern.

Regular expressions define by means of some simple rules patterns of characters, i.e., they describe the order in which characters can follow each other.
Typical token defined by regular expressions are words, numbers, times, data, quotes etc.

6.5 Syntax tree

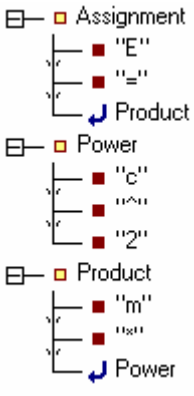
Patterns of tokens can be described by rules similar to those, which describe regular expressions. For example a main clause consists of a sequence of words, followed by a dot, or a table consists of the header with the names of the columns followed by a sequence of rows.

The syntactical analysis shows the context of a token inside of a grammar. This context can be presented as a tree structure.



In this example the tokens are very simple. Each of them only is one character: "E", "m", "*", "c", "^", and "2". In the picture the tokens or terminal symbols are the leaves of the tree, written at the bottom. Graphical these leaves are characterized by the fact, that they only are connected by one single line; grammatically they are indivisible (see remark).
This makes the difference to the other nodes of the tree, which represent so-called non-terminal symbols. Non-terminal symbols can be divided into the terminal symbols. In the graphic, they are starting points of branches.

In the TextTransformer the syntax tree of the picture would be separated into the three structures of the non-terminal symbols and look like:



Remark: That terminal symbols are grammatical indivisible, doesn't mean, that they can't be divided

into characters. In more complex tokens as used for the example, that will be the case.

6.6 Productions or non-terminal symbols

The non-terminal symbols are described by rules, which determine the order of the terminal symbols. Such a rule is called a **Production**. To formulate a production, TETRA has is a script language with a syntax similar to that of the regular expressions. While the elements of the regular expressions are characters, a production describes the concatenation, repetition etc. of token elements. In this regard productions are patterns of patterns.

A production has two tasks. Beneath of the just explained task:

1. to determine, what will be recognized in a text
2. to determine, what will be done with the recognized text

Point 2 denotes the synthesis mentioned above.

Corresponding to the two tasks, there are two kinds of sections inside the definition of a production, which are separated by special brackets. In the example presented on the next page the brackets "{{" and "}}" are used, to separate the semantic actions denoted by point 2 from the syntactical code (point 1).

6.7 Productions as functions

A production may be considered as a specification for creating a routine that parses a part of the input text. By creating code, this specification will result in a real routine. The routine can return a valueand will constitute its own scope for parameters and other local components like variables and constants. These again, can be passed to other productions, which are called like functions inside the body of the first production. The called productions parse sub sections of the part of text, which is parsed by the calling production.

Example

In the following example the die *Outer* production calls the *Inner* production, which returns the string, which the *Inner* production has recognized.

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


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

Input: "a b"
output: found: a and b

Input: "a c"
output: found: a and c

The created code is (in essence):

```
( 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();
  }
```

The scanners are holding the information about the permitted token and the variable *xState* holds the information about the last recognized and the expected token.

Remark: the parenthesis around "b" | "c" are necessary. Otherwise the first alternative would return an undefined value.

6.8 Four uses of productions

It is the main task of the productions to represent the grammar of a text so that it can be parsed with them. However, it is also possible in the TextTransformer to use productions for tasks which are subordinated to this main task.

Altogether, there are four uses of productions in TETRA:

1. as constituents of the **main parser**
2. to the look-ahead in the text
3. for parsing text inclusions
4. as sub-parsers in semantic actions

6.9 Looking-ahead

The decision by which production or branching within a production the analysis of a text has to be continued always depends on the tokens following in the text.
A parser most efficiently works if the next token already makes this decision possible. If a look-ahead of only of a single token always suffices for analyzing the text, then the parser is called **LL(1) conform**. It is an art of the developer to formulate the grammar - the set of the productions - so that the parser gets LL (1)-conform.
The TextTransformer offers a great support at this task since it generates notes and error messages automatically if the grammar is not LL(1). TETRA also permits, however, the **look-ahead of arbitrarily many many tokens**, if this should be required. The already known productions are used for such a look-ahead once more. A production can be applied as a trial to a text to test whether it can parse it or not. The analysis of the text then can depending on the result of this test be continued in a different way. (At the tentative application of productions no semantic actions are executed.)

The look-ahead is explained again concretely at the example of a formal salutation at the beginning of a letter Either it is

```
Dear Mr NAME
```

or

```
Dear Mrs NAME
```

To parse these short texts at first one could have the idea, to formulate the following productions (the character "|" separates alternatives from each other and can be read as "or"):

```
Salutation ::= SalutationBeginning NAME

SalutationBeginning ::=   "Dear" "Mr" NAME
                        | "Dear "Mrs" NAME
```

A look-ahead of two words is required here. After the word "Dear" is recognized the following word decides, which alternative of *SalutationBeginning* has to be chosen and you must go back in front of the word "Dear" again to start with the real processing of the text.

The following productions are better:

```
Salutation ::= "Dear" Gender NAME
Gender ::   "Mr"
          | "Mrs"
```

Here always the next word decides how to continue with the productions.
Astonishingly many texts can be parsed according to this LL(1) principle, if one designs the rules correspondingly.

There nevertheless are cases a look-ahead of only one token doesn't suffice. By the TextTransformer it is possibly to look-ahead arbitrarily far in the text in such cases. E.g. it could be necessary to know already before parsing a sentence whether it is an interrogative sentence or not. However, the interrogative sentence can be identified by the question mark only at the end of the sentence. This could be managed as follows:

```
IF( InterrogativeSentence() )
 InterrogativeSentence
ELSE
 NormalSentence
END

InterrogativeSentence :: InterrogativeSentenceWordOrder "?"
NormalSentence ::= NormalSentenceWordOrder ( "." | "!" )
```

6.10 Inclusions / comments

Comments in programming languages are a typical example of what is meant here by an inclusion. The concept "inclusion" was chosen here to describe the general structure of this example.

Comments or other inclusions can be slid into texts in arbitrary places. The syntactic structure of the texts isn't destroyed by inclusions, even if keyword of the grammar (programming language) are used in them.

Example:

The use of c++ comments is possible in the scripts of the TextTransformer:

Into the variable declaration:

```
int iCount, iEnd;
```

comments can be included without making the variable declaration syntactical invalid

```
int iCount  /* int variable as counterr */, iEnd /* maximum */;
```

C++ comments are sections of text, which are **included** between the two tokens"/*"and"*/". Comments can contain arbitrary text, and therefore the keyword "int" may occur there. However, it isn't interpreted here as a variable type.
Comments were treated exclusively as a part of the ignored characters and recognized by complex regular expressions as whole in older versions of the TextTransformer.
However, it is also possible that the texts of an inclusion aren't arbitrary but that they obey a grammar of their own. E.g. there are conventions, by which C++comments are enriched by instructions which can be extracted as a documentation of the program code.
So besides the parser for the real program code a second parser is needed for the documentation.

It is possible in the TextTransformer to insert productions in a project for this second parser and to let them execute immediately in the change with the main parser. It is even possibly to nest arbitrarily many different parsers in each other. These parsers can operate on sets of tokens of their own respectively so that e.g. the token "int" only is recognized by the code parser, but not by the comment parser.

6.11 Sub-parser

A production can be called directly from the interpreter code. It then is not part of the real grammar of the parser in which this interpreter code is embedded. The called production is rather a start rule for a separate parser and a new input text is passed to it explicitly.

6.12 Family concept

There is a kind of programs, which is similar to the TextTransformer: the so-called parser generators. All of these programs need one superior production, the start rule, by which the parsing of a text begins. A whole project then consists in exactly those productions on which the start rule depends.

In contrast to the other parser generators the set of productions and tokens of a TETRA project is open. From the set of existing productions, each of them can be chosen as start rule. Productions and tokens on which the actual start rule depends will be compiled automatically.

Example

A project can contain a collection of rules to translate a programming language into a different. Ideally, such a collection would be complete. Than there would be a general start rule, which could translate each program of the source language to the target language.
To write all of the productions, which are necessary for a complete translator, is an ambitious task and often not really necessary. It might be more economic for an occasional translation of parts of the source language, only to write an automated translator for certain constructs and to translate the rest manually.
The TextTransformer is a good tool for managing such a pool of rules.

6.13 Tests

In the TextTransformer test scripts can be written, to check that the productions are working correctly. Tests can be helpful, when you create new rules. But primarily they are used, to assert, that the improvement of one part of project not has unexpected consequences in other parts.

A test consists in the isolated execution of single productions. As for a whole project also for a test an input will be transformed to an output. The output will be compared with the expected result. If they are not equivalent an error message will be produced.

Part VII

7 Examples

The examples are constructed like a tutorial. It is suggested that you experiment with these before developing your own applications.

Exchange of words
Conversion of an Atari text
Calculator
Text statistics
GrepUrls
E-mail address
Guard
Bill
XML
Java

Included in the examples are parsers, which are used by the TextTransformer itself:
The semantic code of these examples is removed (except *EditProds*).

TETRA productions
TETRA-EditProds
TETRA interpreter
TETRA import

Cocor import

Additional examples can be found at

http://www.texttransformer.org

There are some videos at

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

7.1 Exchange of words

By this simple example of the exchange of two words some essentials about writing and using of a TETRA project are explained.

Problem definition:

When writing a text it sometimes happens, that inadvertently two expressions are exchanged by mistake. That may be similar names of two persons: Marcuse and Mabuse or two foreign words: ontological and ontic, or the names of two chemical substances N,N-Di-(2-hydroxyethyl)-N',N'-dimethyl-3,7-diaminophenothiazoniumjodid and N,N'-Di-(2-hydroxyethyl)-N,N'-dimethyl-3,7-diaminophenothiazoniumjodid. This exchange by mistake shall be reversed.

Common method of correction:

If you want to correct these mistakes by the method of searching and replacing the words inside of a normal text processing software, you had at first to replace one of the two expressions by a third (e.g.Mabuse by Labuse), then the other expression by the first (Marcuse by Mabuse) and finally the third by the second (Labuse by Marcuse).

TETRA program:

Inside of the TextTransformer an exchange is possible in one run. However you have to write a little program (only one rule). This is worth the cost quickly, if you have to correct several texts. Also you can exchange not only one pair of words but as many pairs as you want.
TETRA only needs one run for all the exchanges, because every time an arbitrary word of the list is found it will be replaced directly by its counterpart. So the text must be processed only once from left to right.

Two versions of the project are presented:

a) A version with only one production and
b) A version with a simplified production and some additional tokens

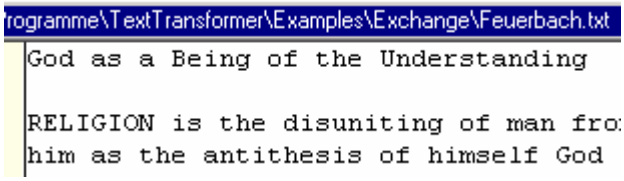
7.1.1 Execution of a project

To test the project, a section of a text from the philosopher Ludwig Feuerbach will be used. It exists in sub-directory of *Examples*:

"\TextTransformer\Examples\Exchange\Feuerbach.txt"

In this text the words "God" and "men" shall be exchanged.
Open the text by the menu *File->Open* and use the line break button:

So you will have a better view of the text. The long lines are broken and the whole text is readable.



The project for the exchange of words is in the same directory:

"\TextTransformer\Examples\Exchange\Exchange.ttp"

Open the project by the menu *File->Open project.* Now, on the right side of the user interface in the tab *Syntax tree* the name of the production: *Exchange appears*.

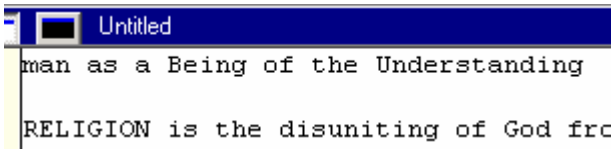
Before it shall be explained, what a production is, you already can execute the program, to see, how a texts can be transformed in the TextTransformer IDE.

Please, click on the button in the tool bar to execute the program:

A progress bar shows the course of the transformation. As the text is very short, the tool bar will disappear very soon and you can see the result of the transformation in the output window.
If you move the mouse cursor to the separating line between the input and output window, the cursor changes its form to
:

While pressing the mouse button, you can size the output window to the same size as the input window. Now click into the output window and activate the line break as you did before in the input window.

In the menu *Options->synchronize* windows you can synchronize the windows. If you scroll one of them, the other will scroll too. So both window are displaying the same section on text.

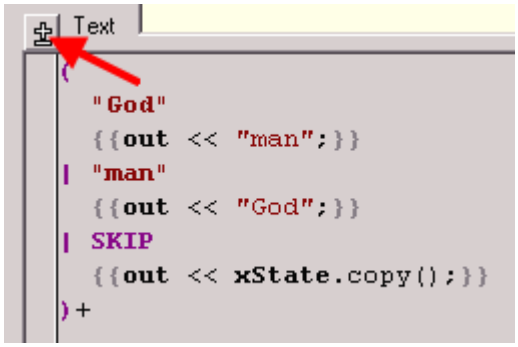


The comparison of the source text and the target text shows, that the materialistic philosophy of

Feuerbach was transformed to a kind of reversed idealism, and this, only by exchange of the two words: "god" and "man".

7.1.2 Production

If you click on the name *Exchange* in the right block of the user interface, in the left block the tab will change automatically to the production page and the properties of the *Exchange* production are displayed. The text of the production is:



The syntax and meaning of the TETRA productions is explained in the chapter scripts in detail. The following explanation gives a first impression of them.

At first only the part of the code shall be discussed, which carries out the analysis of the text and the part, which stands for the remodelling of the text, shall be ignored. Please, click the button marked with the red arrow to collapse the semantic code. Then you get the following picture:

(...)+

The whole rule is included into parenthesis: (...)+. This means, that the source text that the rule describes is a repeat (at least once) of what is described inside of the parenthesis.

'|'

Inside of the parenthesis there are three sections separated by a pipe character '|'. This symbol separates alternatives. The whole text therefore consists of the repeat of three alternatives.

Abstaining from the expression included in the double braces, these are the alternatives:

"God" | "man" | SKIP

That means, at each position of the source text there is either the word "God" or the word "man" or the third alternative, denoted by SKIP, applies.

SKIP

By the key word SKIP TETRA is instructed to skip all the text, which is not an alternative to SKIP; in the current case that is all the text, which not the word "God" or the word "man". Now it's quite logical that the *Exchange* production covers all text. The whole text consists in "God" or "man" or other words.

{{..}}

The parts of the rule included into the double braces of the first picture still have to be explained. They contain the instructions, which have to be executed, as soon as the preceding alternative was recognized.

"out <<"

The instruction "out <<" means, that the following expression shall be written into the output. If the word "God" was recognized, the word "Man" will be written into the target text and reversed if the word "man" was recognized, the word "god" will be written.

xState.copy()

In the case of the SKIP alternative xState.copy() will be written into the output. *xState* represents the actual state of the transformation process, and xState.copy() delivers the last recognized section of text.

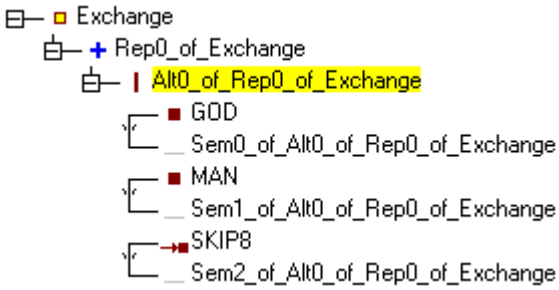
7.1.3 Analysis step by step

The explanation of the rule can be demonstrated in detail, by executing the program step by step.

To do this, first click on the *Tetra*-Tab of the left block in the user interface.

Now you can execute the program step by step by means of the single node button:

After you have clicked the button for the first time, a tree view is opened below of the name of the *Exchange* production, which shows the syntax of the production. After a second click on the button, the tree will look like:



The node *Exchange* represents the whole production.
The node *Rep0_of_Exchange* represents the parenthesis "(...)+"
The node *Alt0_of_Rep0_of_Exchange* represents the set of alternatives.
Each alternative consist of two nodes: a node representing the text, which shall be recognized, and a node representing the semantic action.

At each further step the yellow mark walks to the next tree node, which leads to the recognition of the next section of source text. Every time a node, which represents a semantic action, is left, the corresponding instructions will be executed; that means here: the target text will be extended.

The stepwise execution of a program is carried out in the so-called debug mode, and the TextTransformer has to be set back into the normal mode. This is done with the reset button:

Thereby the output will be deleted too.

7.1.4 Using tokens

The *Exchange* project contains a second production: ***Exchange_Token***.
Exchange_Token is an alternative possibility to program the exchange of words.
In this version the words, which shall be exchanged, are defined on the token page, where they can be combined directly with actions.

On the token page a token **GOD** can be defined as follows:

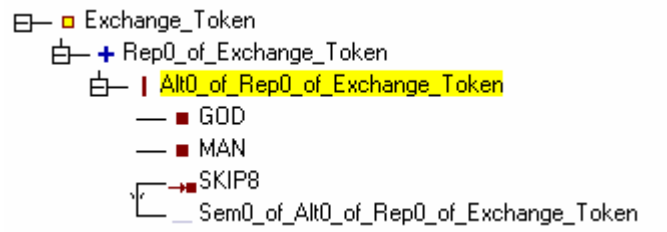
The connected action in the lower field will be executed automatically, as soon as the expression "God" is recognized. With the analogous token for "man" the original production now is simpler:

```
Exchange_Token =
  (
    GOD
  | MAN
  | SKIP
    {{out << xState.str();}}
  )+
```

To execute the new production, you have to go to the *Tetra* page again and you have to choose *Exchange_Token* as start rule in the box of the tool bar:

Than you can use the single node button as explained above.

The presentation of the production in the syntax tree has become now more easy too, because the actions are not depicted any longer_



It is a matter of taste, which version of the exchange project you prefer in such a simple example as presented here. In more complex projects, it can be advantageous, to immediately connect a token with an action. The token then can appear in different rules, but the action remains the same.

7.2 Conversion of an Atari text

The example *Conversion of an Atari text* is similar to the previous one. However, not words, but special characters are exchanged here.

Problem definition:

Texts, which were written on an Atari computer shall be used in a text editor under Windows. A special problem arises if a text uses letters, which are not part of the English alphabet: e.g. the umlauts in the German text example.

TETRA program:

You can find such a text in the subdirectory *Atari* of the *Example* directory of TETRA: "\TextTransformer\Examples\Atari1\Test.txt". Please open it by *File->Open*.

After you have opened the text, its beginning looks like:

```
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,
```

The text contains characters, which are represented by an empty square "□". On the one hand, these are characters which define text attributes (underline, bold, italics) in the Atari word processing. On the other hand, these are special characters like the umlauts, which aren't represented correctly. So the text must be converted.

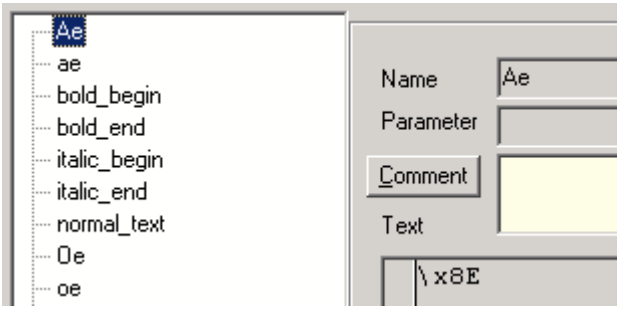
You can load the TETRA project by which such a conversion can be executed, by the menu: *File->Open project*. The Project is in the same directory as the test text:

\TextTransformer\Examples\Atari\Atari.ttp

The tokens, productions and actions of the projekt are presented. Finally a second version of the project is presented, which transforms the text into the RTF format.

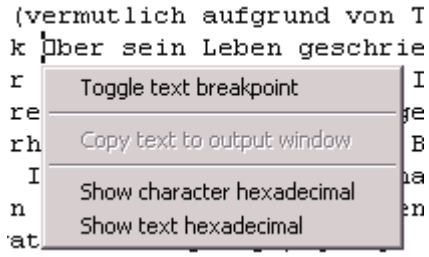
7.2.1 Tokens

To translate not readable characters into readable characters, at first they have be found in the text. The text elements - **tokens** - which the translation program has to search for, have to be defined on the second page of the TETRA program. If you click the **Tokens tab** of the register with the mouse, a list of names of the defined terminal symbols is show on the left side of the page.

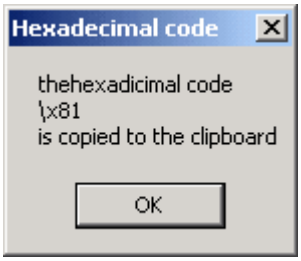


If a name of the list is selected, then the definition of the corresponding symbol appears in the text window. With an exception of one definition all definitions of the Atari project are similar to each other: a backslash '\' followed by an 'x' and two numbers. If e.g. the symbol ***ue*** is selected, then the following expression appears in the text window: ***\x81***. This expression is a number in a hexadecimal notation, which is assigned to the character by the ANSI-set. Instead of this expression also empty square "□" could have been written which occurs in a place of the text, which shall be replaced by a 'ü'. The token text windows then would look the same, however, for all umlauts.

The expression: \x81 can be found easily, if you go back to the editor on the TETRA working page. There you can place the text cursor before the unknown character and click the right mouse button. Now a popup menu appears, where you can select: ***Show hexadecimal character***.



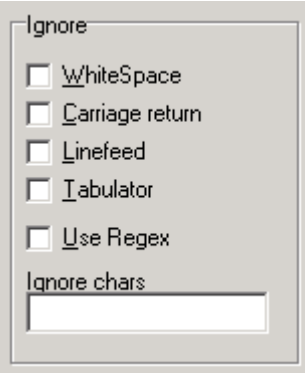
A dialog appears, which shows the hexadecimal expression. The expression is copied into the clipboard automatically and it can be inserted into the definition of a symbol.



One of the symbol definitions is much more complicated than the others:

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

This expression defines the text sections, with no special characters and no text attributes. Inside of the brackets there is a negation symbol '^'. This symbol is followed by a list of all hexadecimal expressions, which are used for the other token definitions. The square brackets are used to define a set of characters, here: the set of all characters, which are no special characters and no text attribute; e.g. a letter of the alphabet or a punctuation mark. The plus sign following the square bracket indicates that at least one character of the set has to occur, but also a sequence of such characters with arbitrary length is allowed.
normal_text normal_text also includes line breaks, tabulators and blanks. These characters per default are ignored in TETRA projects. For the Atari project this standard setting was changed. In the menu *Options->Project options* all ignorable characters are disabled.



7.2.2 Productions

While the Exchange example only used one production, the Atari project consists of three. It would have been possible to use only one production here too, it then would be quite extensive, though. For the following explanations at first the text inside of the brackets "{ ="and "=}" shall be ignored. (see Actions).

The rule ***special_char*** consists of the alternative symbols of the special characters.

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

The alternative relation is expressed by the character '|'.
The rule *special_char* matches the source text at positions where a special character can be found.

The production ***textattribute*** consists of the alternative symbols for the characters by which the Atari text attributes are defined.

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

The production *textattribute* matches the source text at positions where a character can be found, by which the Atari text attributes are defined.

Finally the start production: ***Atari***.

```
(
  special_char
| textattribute
| normal_text
)+
```

Perhaps who has studied the first example attentively, will have noticed the structural similarity between the *normal_text* token of this example and the SKIP node of the Exchange example. The *normal_text* token was actually defined here only for didactic purposes and the production could have been formulated also analogously to the *Exchange* example

```
(
  special_char
| textattribute
| SKIP
)+
```

The ***normal_text*** token describes all text, which doesn't contain any text attribute or special character and the **SKIP** node of the *Exchange* example described all text, which doesn't contain the words "God" or "Man".

7.2.3 Actions

Like in the first example a semantic action is executed after each recognized text section (character), which writes some text into the output.

The text, which is recognized by the *normal_text* (SKIP) token, simply will be copied:

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

Special characters will be translated into readable characters, e.g:

```
| ue   {{ out << "ü"; }}
```

For the text attributes however, there are no semantic actions defined. So these attributes simply are ignored. Every word processing program has its own way to code these attributes in its text documents. As an example a conversion into the RTF format is discussed below.

What shall be made with the recognized sections of text, is written in the productions inside of the pair of brackets "{=" and =}". Here the so-called semantic actions are defined. In the tree view they are represented by nodes, the names which of begin with "Sem"; e.g. *Sem0_of_Alt0_of_Rep0_of_Atari* .

The instructions for the semantic actions are a subset of the programming language c++. For the *Atari* project only one instruction is needed: the *shifting* of a text into the output.

For example, the production ***special_char*** contains the line:

```
| ue   {= out << "ü"; =}
```

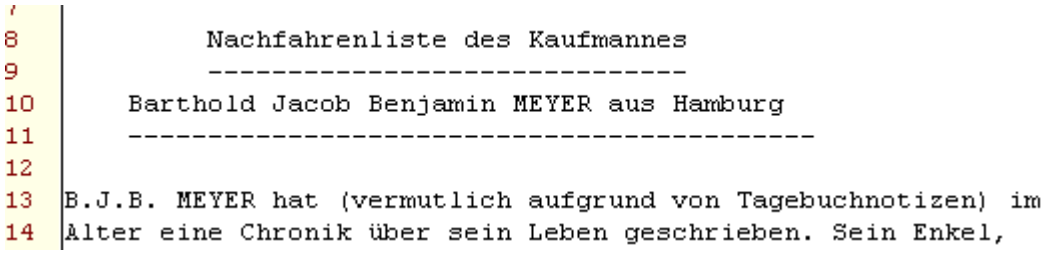
This means: as soon as the symbol *ue* is recognized, execute the action

```
out << "ü";
```

that means append the text "ü" at the target text. The semantic actions, which are executed after the recognition of the other tokens, are similar.

For the text attributes however, there are no semantic actions defined. So these attributes simply are ignored. Every word processing program has its own way to code these attributes in its text documents. As an example a conversion into the RTF format is discussed below.

If you now execute the program, the part of text shown at the beginning now looks:



The text attribute characters are removed and an "ü" replaced the empty square "□".

7.2.4 Conversion into RTF

Until now, the characters, which represent the Atari text attributes: underline, bold and italic, simply were ignored, because plain texts cannot contain such attributes. An example for a document, which can contain such attributes, is an RTF-file, which uses the Rich Text Format: RTF.
The project

...\Tetra\Examples\Atari\Atari2Rtf.ttp

demonstrates, how the Atari text can be transformed into the Rich Text Format. The relevant parts of the RTF specification are immediately used here, without explaining this specification in detail. A good introduction is "RTF Pocket Guide" of Sean M. Burke.

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

The important first chapter of the book as a PDF file can be downloaded there. The original Rich text specification can be found at:

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

According to RTF specification, parts of text with special attributes are enclosed into bracket "{" and "}" and at the beginning of the enclosed text the attribute is written:

italic: \i
bold: \b
underline: \ul

So the token actions for e.g. italic become to:

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

In RTF the umlauts have to be represented as a backslash followed by an apostroph and the hexadecimal code of the character. So the token action e.g. of *ae* becomes:

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

Line breaks, which shall be shown in the text-processing program, cannot be written directly into RTF-files. They have to be coded by "\line" or "\par". So an additional token has to be defined, which recognizes line breaks:

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

(By *endl* a line break is written into the RTF document to make a logical partition into the RTF source.)

The whole RTF file must be enclosed into the brackets "{" and "}" and the first bracket must be followed by a header, which specifies the RTF version, the fonts used and the font size. So the begin of a RTF file looks like:

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

To write this header, we will use the page for class elements of the TextTransformer for the first time. There you can variables and functions, which then can be used inside of the whole project. For the Atari project three functions are defined:

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 and *RtfFont* could have been combined into a single function too or even written directly into the text of the Atari production. But by the division into these functions, the logical structure of the whole project becomes clearer.

The Atari production now is:

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

After you have transformed a text, you have to save the output as an RTF-file. That means, the extension of the file must be "rtf". Now this file can be opened into a word processing program - e.g. WordPad -, which is able to display the RTF format. There you can see for example *Benjamin* now correctly written in italic

Jacob *Benjamin* MEYER

7.3 Calculator

You should know the most essential operation elements of TETRA.

Problem definition:

Arithmetical problems like "(3.2 + 8.9 - 4.6) * 5.6" shall be calculated.

TETRA Program:

This is a classical application of a parser. The numbers, operators and parenthesis have to be extracted from the input to calculate the result. The project consists of several tokens and five productions, which will be explained on the next pages.

This project again exists in **two versions**:

a) The first version demonstrates the use of parameter references and

\TextTransformer\Examples\Calculator\Calculator1.tpp

b) A second uses return values

\TextTransformer\Examples\Calculator\Calculator2.tpp

7.3.1 Tokens

The tokens for the operators and the parenthesis are defined directly inside of the productions.

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

A complex token for the recognition of numbers is defined as a regular expression on the token page:

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

"\d" a single digit
"\d+" a repeat of digits, at least one
"\." the dot (without backslash '\' the dot has a special meaning inside of a regular expression).
"\d*" a repeat of digits, or no digit
"?" an optional occurrence of the content of the preceding parenthesis
"|" an alternative

So a number is defined either as a sequence of digits, optional followed by a dot followed by null or more digits (e.g: "123" or "123." or "123.45"). Or a number begins with a dot followed by a sequence of digits (e.g.: ".45")

7.3.2 Production: Calculator1

The start rule for the Calculator1 program is:

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

In this example the actions are included into the double braces **{{...}}**. For such actions special project options are valid. These options determine, whether the actions are interpretable inside of the TextTransformer or not. For this project the default value is maintained, that means: they are interpretable.

The action:

```
double d;
```

declares a variable of the type double, which shall take the result of the calculation. Variables of this type can contain numbers with fractional digits.

The action:

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

already is know from the previous examples. The value of the variable followed by a line break will be written into the output.

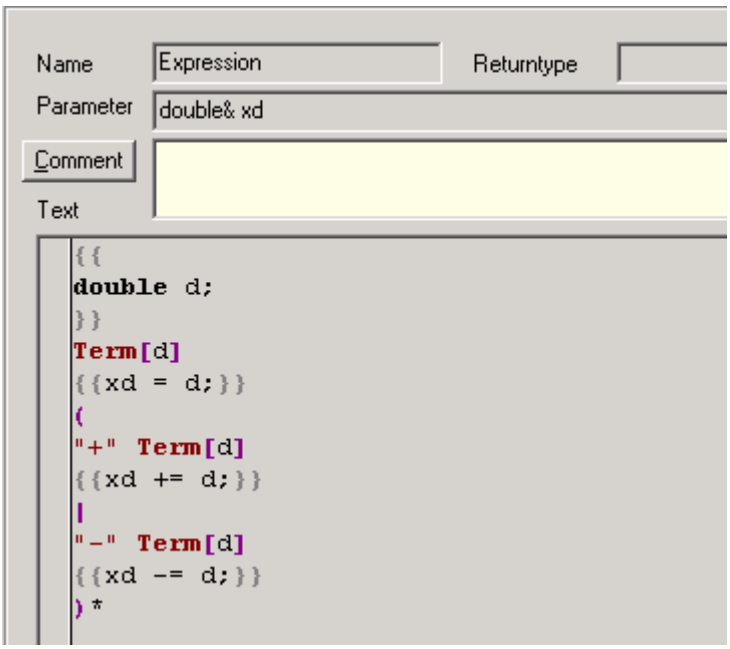
Between the just explained two actions the production *Expression* is executed.
Inside of this production, the result is calculated. By means of the bracket "[d]" in

```
Expression[d]
```

the variable d is passed to the *Expression* production, where it can get its value.

7.3.3 Production: Expression

The production: *Expression* has a parameter:



Parameter: `double& xd`

This is the interface, where the variable, defined in the *Calculator1* production, is put into the *Expression* production. Inside of the *Expression* production it has the new name *xd*. **double** again is the type of the variable. The "&" characterizes the variable as a reference, that means, that the value, which may be changed inside of the *Expression* production will accessible afterwards outside in the calling production (Calculator1). Without the "&" the variable would keep its value inside of the *Calculator1* production, even if the value of xd inside of the Expression variable had changed.

Leaving out the actions, the rule simplifies to:

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

An expression is a term, to which an arbitrary number of other terms can be added or subtracted.

The instruction:

```
double d;
```

declares (as above) a new variable of the type double, which shall take the value of the term.
The result of the first term will be assigned to the reference variable xd:

```
xd = d;
```

The values of the following terms will be added or subtracted:

`xd += d;` respectively `xd -= d;`

These expressions are a shorter notation for:

`xd = xd + d;` respectively `xd = xd - d;`

7.3.4 Productions: Term and Factor

The ***Term***-Production is constructed analogously to the *Expression* production; multiplication and division take the places of the addition and subtraction:

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

If the denominator is null, the program is stopped by a call of:

throw `CTT_Error("Division by null");`

The text "Division by null" then will be shown in the log window..

The production ***Factor*** is:

```
{{
bool bPlus = true;
double d;
}}
(
"-"
{{bPlus = false;}}
)?
(
Number[d]
|
"(" Expression[d] ")"
```

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

It can be shown without actions by clicking on the collapse code button:

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

Parenthesis followed by a question mark characterizes the included expression as optional, i.e., it may occur in the text or not.
So a *Factor* is an optional minus sign followed by either a number or a bracket.
Here the parser becomes **reflexive**. The *Expression* production had called the *Term* production, which called the *Factor* production. Inside the *Factor* production the *Expression* production can be called again. This reflexivity mirrors the possible nesting of parenthesis inside of arithmetic problems, e.g. "3 + ((3.2 + 8.9 - 4.6) * 5.6)"

The minus sign is optional. If it exists or not in the text, will be memorized in a boolean variable. Variables of the type **bool** can have the value *true* or *false*. The variable **bPlus** is declared at the beginning of the *Factor* production and set to the value true:

```
bool bPlus = true;
```

If the minus sign is found at the actual text position, this value will be set to *false*. After the evaluation of the *Number* production or of a bracket, the result will be negated or not, depending on the value of ***bPlus***. For this are the instructions:

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

The structure

```
if ( <condition> )
  <instruction1>;
 else
  <instruction2>;
```

means, that if the condition "condition" is fulfilled, that means, it has the value *true*, the instruction "instruction1" will be executed otherwise "instruction2".

7.3.5 Production: Number

The *Number* production only consists of the terminal symbol **number** (::= "\d+(\.\d*)?|\.\d+", see above) and an action, which translates the text, matched by the symbol, into a number:

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

This translation is made by a special function **stod**. **stod** can be read as an abbreviation of "string to double". A **string** is passed to the function and it returns a **double**-value. To memorize: xState.str() returns the text of the last recognized token. So **stod** makes for example the value 123.45 out of the text "123.45".

7.3.6 Return values

The intermediate results of the calculator till now were delivered from reference variables from the invoked productions. Reference variables have the advantage, that they can be used in arbitrary number. In the calculator example, there always are only single intermediate results.

The calculator example using return values, is in the directory:

\TextTransformer\Examples\Calculator2

The start rule now is shortened to:

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

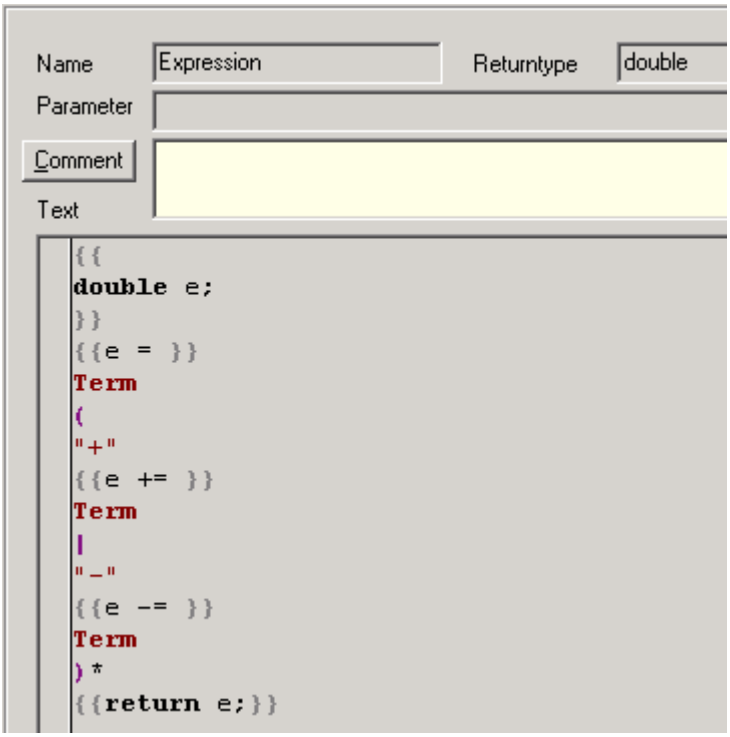
The instruction for the output in the first line is incomplete according to the pure syntax of c++. Inside of the TextTransformer this notation is allowed, if immediately after the shift operator and the closing braces of the semantic action a call to a production follows. The called production must return a value - see below -, which can be written into the output.

The finishing action is:

```
out << endl;
```

endl is another notation for "\n", that means for a line break.

The *Expression* production in Calculator2 has no parameters but the **return type: double**



A production (or token) with a return type must return a value of this type. This happens in the last line:

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

The **double** variable **e** is declared at the begin of the production. Its value gets the variable from the *Term* production in three lines, which again are shortened c++ instructions:

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

The *Term*- and the *Factor* production are rewritten in the same manner as the *Expression* production, so that they use a return type instead of reference variables.

In the ***Number*** production the declaration of a temporary variable can be renounced, by the fact, that the return value of the **stod** function is passed on directly.

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

7.4 Text statistics

You should know the most essential operation elements of TETRA.

Problem definition:

The number of lines, characters, word and sentences of a text shall be counted.

TETRA Program:

The project is in:

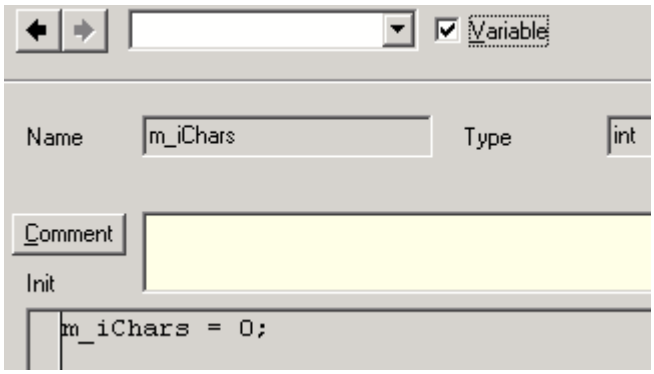
\TextTransformer\Examples\TextStats

This project demonstrates the use of class variables and member functions. Beneath its pure linguistic possibilities of application, the program is of practical use, if you have to fulfill some constraints for texts, e.g. when registering a shareware program at a distributor.

7.4.1 Class members

In the previous examples **locale variables** were used, which are declared inside of an action and can be passed to other productions or tokens as a parameter. In this project **class variables** are used. The declaration of a class or member variable is made on a special page of the TETRA user interface. Such variables can be accessed immediately inside of each action of all productions and tokens (and member function see below). The value of a member variable can be changed in one action and immediately is in disposal of other parts of the program.
In the actual project for example there is an integer member variable *m_iChars*, which stores the number of recognized characters. In each token action this variable will be augmented by the number of characters, which is recognized by the token.

The declaration of *m_iChars* is done on the element page.



To declare a variable, you have to enable the according check box in the toolbar. Otherwise *m_iChars* would be interpreted as the name of a member function.
The type of the variable is set to *int* in the *Type* field.

The text field now is entitled: *Init*. For variables this field may remain empty, but it also can be used, to initialize the variable. The initialization is done every time a new text transformation starts.
Here *m_iChars* gets the value 0. This is the default value, which the variable would have obtained too without explicit initialization. But to initialize the variable is a better programming style. If code for a parser class is exported the lack of an initialization can lead to errors, because in c++ integer variables don't get an initial value automatically.

Other member variables to count lines, words etc. are declared in the same manner as *m_iChars*.

A special case is *m_mAbbr*. This variable has the type mstrstr and is initialized by a number of expressions, which are abbreviations, if they are followed by a dot.

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

This list is needed, when counting sentences to distinguish dots, which are marking the end of a sentence and dots, which belong to an abbreviation. Because the list by no means is complete, the counting of sentences will remain uncertain.
Each of the elements of the list is assigned an empty string. These values aren't needed. In this project only the property of the map is used, that searching for a key is very quick, because the keys are in sorted order automatically.

Further there is a member function *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"
```

It prints the formatted result of the counting at the end of the program.

7.4.2 Token

In the project options all ignorable characters are deactivated. So the set of token must recognize all parts of a text, linefeeds and spaces included.

So a text consists of

WORD	words
NUMBER	numbers
ABBREVIATION	abbreviations
CONTINUATION	sequences of dots like "..."
LINEFEED	linefeeds

SENTENCE_END	ends of sentences (dot, exclamation and question mark)
SPECIAL_CHAR	the rest of characters

In the actions of the tokens the counter are actualized. For example the WORD action:

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

Here the counter for words is augmented by one and the counter for characters is augmented by the number of characters, of which the word consists.

A little bit more complicated is the action of the token *ABBREVIATION*: (\w+)\.

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

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

If the recognized text consists of a single letter followed by a dot or if the text preceding the dot is found in the list of abbreviations, the recognized text is interpreted as an abbreviation. Otherwise the dot marks the end of a sentence and the sentence counter is incremented.

7.4.3 Productions

The start rule of the project is *TextStat*. *TextStat* calls in a loop the production *Text*, which consists of the alternative tokens. Both productions are very simple and there is nothing new to explain.

There is an additional production *CountChars*. *CountChars* offers a very simple possibility to count only the characters of a text.

7.5 GrepUrls

You should know the most essential operation elements of TETRA.

Problem definition:

In html files all links shall be found and listed, which refer into the web.

TETRA Program:

The project is in:

\TextTransformer\Examples\GrepUrls

This project demonstrates the use of the TextTransformers as a GREP tool. GREP (Global Regular Expression Print) is in the Unix world a well-known program, used to look for text patterns in files. By means of the transformation manger the TextTransformer can handle such tasks too. This example also demonstrates how the found information can be stored in containers and finally formatted before printing.

7.5.1 Productions

If you look at the text of an html page, which normally is show in the browser, the following for example is a link to the pages of the TextTransformer:

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

If the text enclosed into quotes - the URL - begins with "http://www.", the link refers to an external web page. Exactly that is, what shall be found here. Two simple productions fulfill the purpose :

```
GrepUrls ::=

  (
    Url
  | SKIP
  )*

Url ::=

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

A html page consists of the desired URL's and other text. The URL is introduced by "<a href=\"http://www." and extends up to the next quote mark.

The case-sensitivity is deactivated in the project options, as HTML is case insensitive. So tags like

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

are found too.

7.5.2 Member variables and methods

Now some variables and methods shall be presented, which are defined on the element page.
The positions of the URL's in the texts shall be collected in a member variable of the type *mstrstr*:

```
mstrstr m_mUrl
```

The key shall be the found URL itself, and the value will be constructed out of the name of the actual file and the according line number. These can be obtained from methods of the parser:

SourceName() name of the actual file
xState.Line() line number

A variable of the type format shall help, to combine the names and the numbers into one string:

```
format m_fPosition
```

By the command:

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

m_fPosition is initialized with the formatting string " Page: %|1$|%|50t|Line: %|2$|".
"%|1$|" represents the position of the first argument,%|2$| represents the second argument and by %|50t| the preceding part of text is padded with spaces to a length of 50 characters. If e.g the file name is "D:\C_biblio\boost\index.htm" and the line number is *52*, and both are passed to the format object by means of the %-operator, this object returns:

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

To avoid very long file names, they shall be expressed in a shorter form with relative paths. For this a special method is defined in the project, which transforms the absolute paths, which are obtained by SourceName() to relative paths:

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

This function consists in only one line. What happens here becomes clear, when it is separated into several steps:

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

The resulting relative file name begins with "..", followed by the part of the absolute file name, which follows on the string, which designates the source directory. For "D:\C_biblio\boost\index.htm" and the source directory "D:\C_biblio\boost" this results in:

..\index.htm

A further method defined on the element page is *AddPosition*, which is called with the parameters of an URL *xsUrl* and the position *xsWhere*:

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

This method takes into account, that there may be several positions for the same URL. If the URL hasn't been found yet the position will be set as value of the URL key in the else-branch. If the URL has been found before, the new position text is simply appended to the old value.

Before the program ends, m_mUrl will be printed by the function *PrintAll.* Hereby all internet addresses will appear in alphabetical order automatically:

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

7.5.3 Put together

The complete code of the Url production is:

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

After the text of the URL was recognized by SKIP, it is passed to the *AddPosition* class method, together with the information about the position, where the address was found.

And finally the complete *GrepUrls* production:

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

An important point still has to be added. The program shall print the sorted list of the Internet addresses, which were found in all files. But the sorting is only possible after **all** files were processed. Because of this, the *PrintAll* function is executed only, if this condition is fulfilled. Whether the condition is fulfilled, can be obtained from the method *IsLastFile* of the parse state class. *IsLastFile* only returns true, if no further file will follow.

7.5.4 Search in whole directory

As usual you can load a html file into the input window of the TextTransformer, to process it. But now it shall be demonstrated, how to work with the files of a whole directory. In our case, these would be all documents of a web site.

To do this the transformation manager is invoked. Most comfortably this is done by the button:

If the project isn't compiled yet, it will compile now automatically, before the transformation manager dialog opens. How to work with the transformation manager is explained in detail in an own chapter of this help. Here only the most important steps to make a management are presented briefly.

You also can find the ready management in:

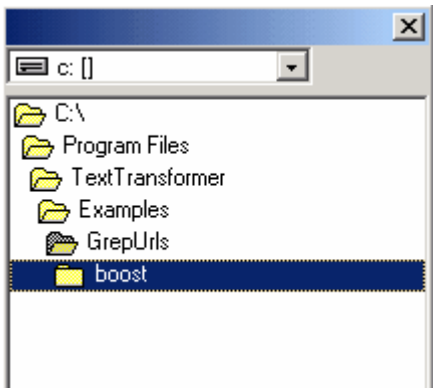
C:\Program Files\TextTransformer.113b\Examples\GrepUrls\GrepUrls.ttm

If you intend in future, too, to work on HTML pages, it is advisable to define a filter just now for this file type. This filter then can be used again and again at the choice of source files in the TextTransformer.

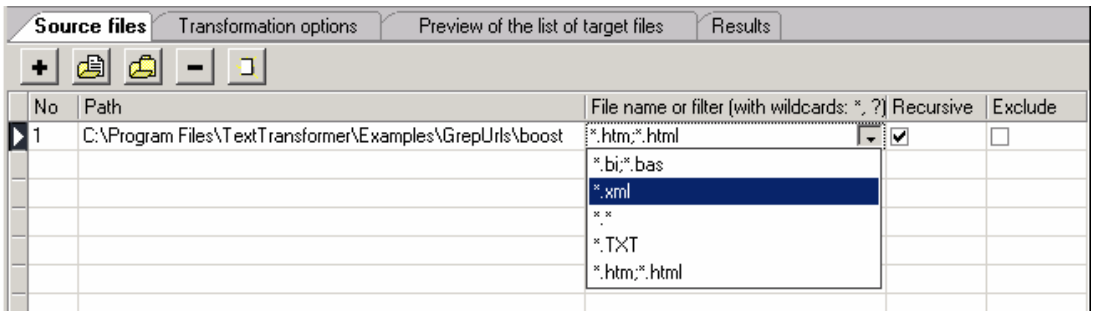
First the folder with the source files has to be chosen. For this example some of the html files from the web site of boost were copied into the UrlGrep directory:

By the button

you can open a dialog to chose the target directory:



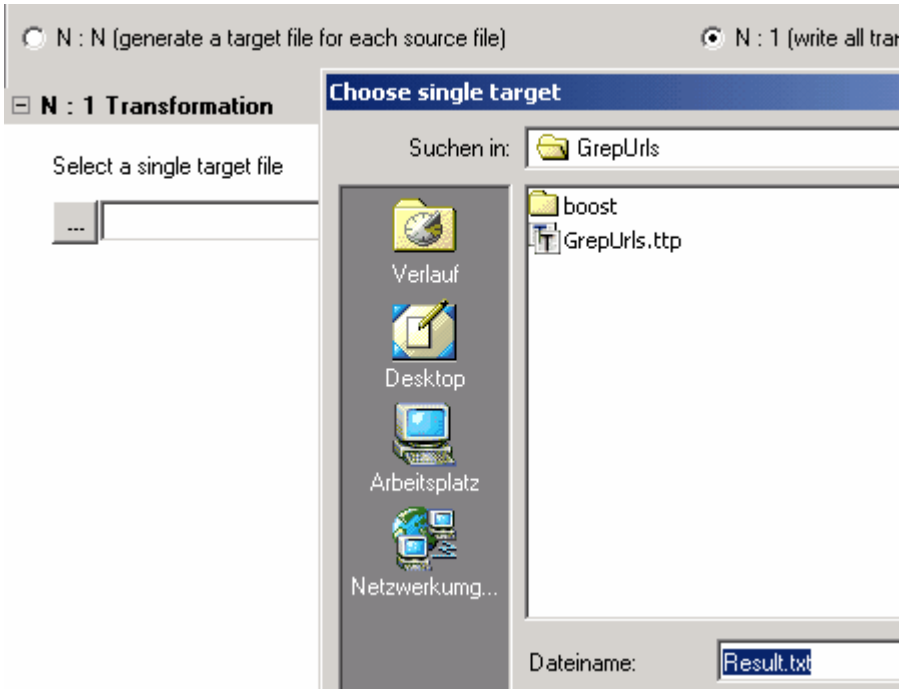
After the choice was confirmed, a new row is inserted in the table of the source files/folders. The *recursively* box can be activated now in this row to search the files of the sub-folders too. If, as recommended above, the filter was defined for HTML files, this can be selected in the choice box of the column *file name or filter* now. However, the filter "*.htm; *.html" can be written also directly to the field.



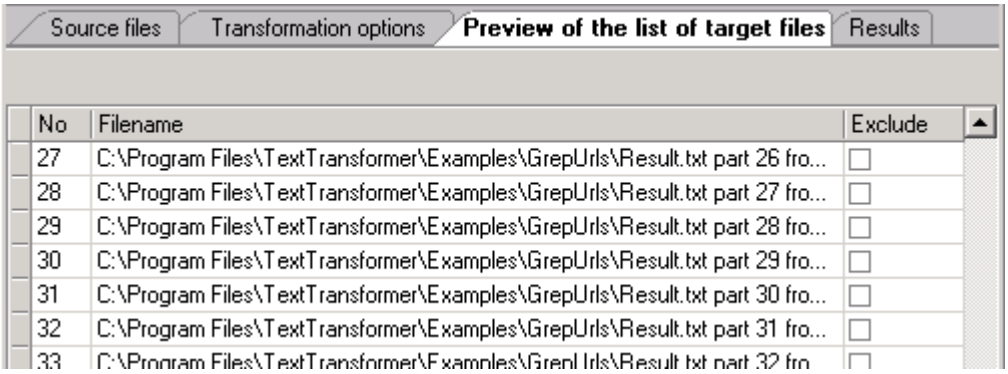
On the *Transformation-Options* page of the transformation manager you have to set now, that a N:1 transformation is planned. That means, that all results of the transformations of the source files shall be written into a single target file.

By the button

then a dialog is opened, by which you can navigate to the desired target folder. Then you can either select an existing file a s target or input a new filename.



A preview is shown on the next page of the transformation manager now, where you can see in the different rows of a table, which source files can contribute to the file to be produced.



Now you can start the search for the addresses:

By a double click on an arbitrary row in the table on the *reult page*, the transformation manager will be closed and the resulting text is shown in the output window of the IDE:

7.6 BinaryCheck

You should know the most essential operation elements of TETRA.

Problem definition:

It shall be tested, whether a file includes binary data.

TETRA Program:

The project is in:

\TextTransformer\Examples\GrepUrls

The example demonstrates the rather simplest parser for binary files and uses a look-ahead production.

The project can be used for other projects as preprocessor which guarantees that the source file isn't binary.

7.6.1 Look-ahead

The project consists of only two productions. One of them, *IsBinary,* is extremely simple:

```
SKIP? NULL
```

It can parse a string only successfully when it ends with a binary null.

This production is used for the look-ahead within the production *BinaryCheck*:

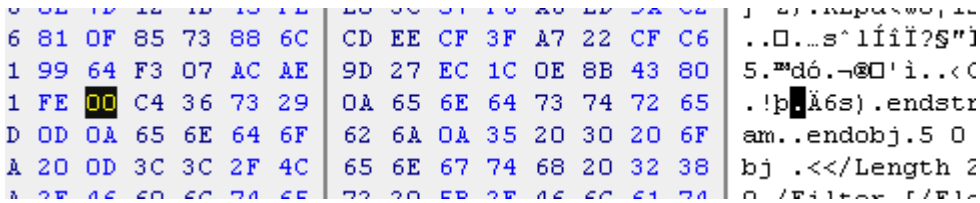
```
IF(!IsBinary())
  SKIP? {{ out << xState.str();  }}
END
```

The difference of the use as a look-ahead compared with a normal call of a production is indicated syntactically by the appended parenthesis "( )" within the IF brackets.
The source file is processed from the current position on -- here the start of the file -- as long as either the production is finished successfully or till a fault appears. *IsBinary* is successful exactly in the case, that there is a null character in the file.

You can see this in the debugger. As the project was opened the file *BinaryTest.pdf* should have been loaded into the viewer too. In the hexadecimal mode of the viewer one can see the null characters. If you step into the look-ahead with

and press the button several times, finally the following picture arises, in which the found null

character is highlighted:



The file is regarded here as a text file if there isn't any null character. (The file could actually nevertheless be designed for a binary use.) If it is a text file, *SKIP*? jumps to the end of the file and the complete text then is written into the output.
So the project can be used for other projects as preprocessor which guarantees that the source file isn't binary.

In addition the size of the file is checked at the beginning of *BinaryCheck*. Binary files often are are very big, because they may contain graphical data or even voice recordings and films. As a limiting value 1000000 bytes are chosen here.

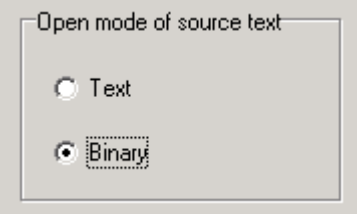
```
{{
int iMaxSize = 1000000;
if( file_size(SourceName()) > iMaxSize )
  throw CTT_Error("file size > " + itos(iMaxSize));
}}
```

Text files seldom achieve this size. Whether even bigger files shall be allowed depends on the kind of the files. You also should take into account that the source files are loaded completely into the working memory for parsing.

7.6.2 Use as preprocessor

One TextTransformer project can be used for a second project as a preprocessor. I.e. the second project takes the target text the first as a source text. *BinaryCheck* would be a suitable preprocessor for projects which shall be applied to all texts of a folder notwithstanding their special file extension. For text files only *BinaryCheck* delivers the source text, at binary files *BinaryCheck* fails and doesn't deliver any text.

The source file has to be opened in the binary mode. Otherwise it would not be guaranteed that BinaryCheck works. If binary files are opened in the text mode, they are read incompletely in most cases.
The binary mode can be set in the project options on the encoding page:



Since TextTransformer 1.5.5 the binary mode is predefined for new projects.

BinaryCheck can be selected in the project options of another project as preprocessor now. You can see a part of the result of a test with *TextStats* in the transformation-manager below:

	28.08.2008	14:39:45	Starting C:\Program Files\Minimal Website\bin\mini_ws.exe
	28.08.2008	14:39:45	file size > 1000000
	28.08.2008	14:39:45	Error on preprocessing with: BinaryCheck
	28.08.2008	14:39:45	C:\Program Files\Minimal Website\bin\mini_ws.exe : Transformation failure
	28.08.2008	14:39:45	Starting C:\Program Files\Minimal Website\bin\readme.txt
	28.08.2008	14:39:45	successfully transformed : C:\Program Files\Minimal Website\bin\readme.txt
	28.08.2008	14:39:45	Starting C:\Program Files\Minimal Website\unins000.dat
	28.08.2008	14:39:46	Error on preprocessing with: BinaryCheck
	28.08.2008	14:39:46	C:\Program Files\Minimal Website\unins000.dat : Transformation failure

You can see that the transformation of *mini_ws.exe* has failed because of his size and *unins000.dat* was identified as a binary file. The text file *readme.txt*, however, was processed correctly.

7.7 E-mail address

You should know the most essential operation elements of TETRA.

Problem definition:

An e-mail address shall be analyzed according to the complex RFC 822 standard.
This standard not only allows simple addresses like:

dme@TextTransformer.de

but also constructs like:

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

TETRA Program:

The project is in:

\TextTransformer\Examples\Mailbox

This project demonstrates how a TextTransformer program can be created from an existing syntax specification. It will be shown too, how hidden LL(1) conflicts can be solved.

There is a complete MIME parser for e-mails at

http://www.texttransformer.org

7.7.1 Syntax specification

Following syntax specification can be found in the book: J.E.F. Friedl: Reguläre Ausdrücke, O'Reilly, 1998.

	Element	Description
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	(a character except specials, space or ctl)+
11	quoted-string	" (qtext | quoted-pair)* "
12	domain-ref	atom
13	domain-literal	(dtext | quoted-pair)*
14	char	An ASCII character (octal 000-177)
15	ctl	An ASCII control character (octal 000-037)
16	space	Space (ASCII 040)
17	CR	Carriage Return (ASCII 015)
18	specials	One of the characters ()<>@,;:\\"\.\[\]
19	qtext	A char except *****, **** or CR
20	dtext	A char except [,], \ or CR
21	quoted -pair	\ char
22	comment	**(** (ctext | quoted-pair | comment)* **)**
23	ctext	A char except '(', ')', '****' or CR

From this specification Friedl constructs a single regular expression consisting in 4724 characters. The following reconstruction of this grammar in the TextTransformer has approximately the length of

specification itself.

7.7.2 Productions and token

Fortunately the syntax of the specification is very similar to the syntax of the TextTransformer. The elements can be transformed easily into productions and tokens. Because hyphens are not allowed in script names of the TextTransformer, they are transformed to underscores. Single characters are enclosed in double quotes. Elements, which consist in single characters or character classes, become tokens. Their names are written in upper case to separate them more clearly from the productions.

	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]

The *comment* production is set in the project options as an inclusion. So, in front of every new token is tested automatically, whether there might be an included comment. Such comments can be nested unlike the use of regular expressions to the comment recognition.

The complete project is in:

\TextTransformer\Examples\Mailbox\mailbox1.ttp

7.7.3 Detecting a conflict

So far no problems occurred and the project seems to be ready. But if you select the start rule *mailbox* and parse it two errors are shown:

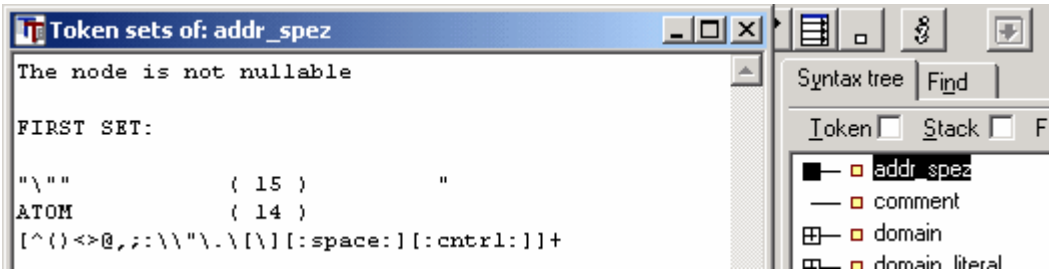
mailbox: LL(1) Error: "\"" is the start of several alternatives
mailbox: LL(1) Error: "ATOM" is the start of several alternatives

If such a LL(1)-conflict exists, the actual text don't suffices for the parser to decide, which rule should be applied next.
The TextTransformer - in contrast to some other related tools - helps to discover such errors. But to remove them, you need some understanding of the grammar, you are developing and some power of deduction.

In the *mailbox* production:

```
addr_spez | phrase route_addr
```

the alternatives are beginning with *addr_spez* and *phrase*. If you select *addr_spez* in the navigation tree at the right side of the screen and click the right mouse button, the first set of this production can be shown:



If you do the same with the *phrase* production, you will see the same First set.
If you look at the definitions of these productions, the cause of the conflict still isn't clear:

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

phrase ::= word+
```

But if you go down another step, the conflict becomes obvious:

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

word ::= ATOM | quoted_string
```

Indirectly as well *addr_spez* as *phrase* are beginning with word.

7.7.4 Solving the conflict

To solve the conflict, *word* must be factored out.
For *phrase* this is easy. The rule will be redefined as:

```
phrase = word*
```

and at all positions, where *phrase* is used in the grammar, phrase is replaced by:

```
word phrase
```

The *mailbox* production just is the only occurrence of *phrase*; so it becomes to:

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

Analogously is dealt with *local_part*. The production is redefined to

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

and all places, where *local_part* exists in the grammar are replaced by:

```
word local_part
```

So addr_spez becomes to:

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

Finally *word* is extracted from *addr_spez* too, whereby this production regains it's old form (but with a new defined *local_part*):

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

Here you must pay attention, because *addr_spez* is also used in *route_addr*.

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

And finally

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

Now the factoring out of *word* can be completed:

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

If you compile this rule now, there is no conflict any more.

The corrected project is:

\TextTransformer\Examples\Mailbox\mailbox2.ttp

7.8 Guard

You should know the most essential operation elements of TETRA.

This example shall demonstrate the use of sub-expression in regular expressions.
The access of sub-expressions will be possible in the future only in the standard and professional version of the TextTransformer.

Problem definition:

Into a c++ source file at the entering and exit positions of methods - i.e. after the opening brace of the function body and before the closing brace - additional instructions shall be inserted, which can be used for debugging or profiling purposes. The name of the class and the name of the function shall be passed to the inserted code.
For example, if there is defined a method:

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

this shall be transformed to:

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

For example the class CGuard can block certain resources in its constructor or initiate a time measurement and leave the resources in the destructor or stop the time respectively.
Another possibility would be the insertion of a try catch block

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

TETRA Program:

The example *Guard* is in:

\TextTransformer\Examples\Guard\guard.ttp

The program is for demonstration only. It is not guaranteed, that it can transform every c++ source file. But it will transform the source files TETRA creates. So two parsers generated by the TextTransformers are used as examples for the input.

\TextTransformer\Examples\Guard\calculatorparser.cpp
\TextTransformer\Examples\Guard\guardparser.cpp

7.8.1 Startrule: guard

The start rule of the program has the name of the project: guard. It shall be able, to process a complete c++ source file, produced by TETRA:

```
(
  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(); }}
)+
```

The names of the alternative productions, which can be called inside of the loop, denote the structures, which shall be recognized by the productions. So the c++ source consists of constructors, destructors member functions, comments, lines, which are processed by the preprocessor, using directives and an undefined rest, which will be recognized by the SKIP symbol. The constructors, destructors and member functions shall be instrumented by "guards". The other parts of the source code shall be copied without change.

7.8.2 Copying source text

The method for copying the text which was recognized by the last token is already known.
The spaces in front of the text shall be included in this copy. I.e. the text of the end of the second to the last token is copied until the beginning of the next expected token:

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

This command is an abbreviated notation for:

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

where the ignored text is accessed by ***xState****.str(-1)* and the recognized text is accessed by ***xState*** *.str()* By means of a parameter certain sections of the input text, relatively to the last recognized section, can be accessed. The parameter "-1" has the special meaning, that it is related to the ignorable characters, which were skipped, when reading the input. xState.str(-1) returns the text between the end of the previous recognized token and the beginning of the last recognized token.
In the "Exchange" examples, there were no ignorable characters. There only were the sections of text recognized by the SKIP symbol and the replaced parts of text.
In the project options of the present example the line break, carriage return, white space and tab are set ignorable (this is by default).
By the call of ***xState****.copy()* after each recognized token is made sure, that the source text is copied to the output completely, the ignorable characters included.

An analogous method for copying the complete text which was recognized by a production is new here:

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

The "lp" of "lp_copy" is for "last production".

7.8.3 Tokens

Besides some token directly defined inside of the productions and the STRING token, in this example there are two groups of token. The first consist of:

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

While they begin differently, they all end with **[^\r\n]***. The last expression describes an arbitrary repeat of characters, which are not line endings. So the group of token describe sections of text, beginning with "//" or "#"" or "using" and extending to the end of the line. A c++ programmer recognizes immediately, that line comments, preprocessor directives and using directives are described.

The tokens of the **second group consist in:**

DECLARATOR
DESTRUCTOR

They are beginning with an expression similar to:

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

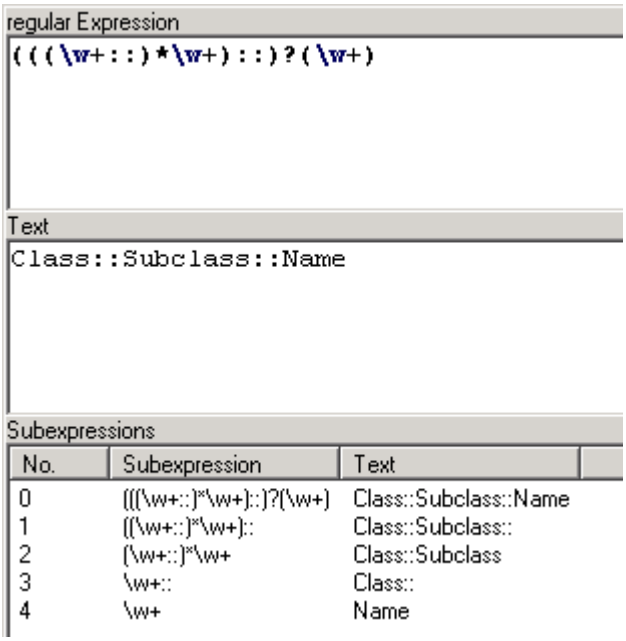
This expression may appear unnecessarily complicated at first. The simple expression:

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

also would recognize texts like:

Name
Class::Name
Class::Subclass::Name
etc.

that are names and class methods.
But the complicated form of the expression allows the access of sub-expressions. Each pair of parenthesis matches a section of the text, matched by the expression as a whole. This section can be accessed in the interpreter. Which section of text is related to which parenthesis can be displayed by a tool integrated in the TextTransformer: menu: *Help->**Regex test.***



From the lower table you can see, that the sub-expression with the index 2 matches the scope, and the sub-expression with the index 4 the name of a class method. This is used in the guard project, to write these strings into different member variables:

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

These variables are defined on the element page of the IDE. They are used in the functions *print_at_enter* and *print_at_exit*, described later.

The complete definitions are:

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

DECLARATOR ::=
(((\w+::)*\w+)::)(\w+) \// scope(s) and name, e.g.: CSub::CClass::Func
\s* \// optional spaces
\([^)]*\) // parameter, e.g..: (int xi)

Notice, that complex expressions can be written and commented into different lines.

Although the tokens are quite similar, TETRA can decide which token matches the text best:

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

will be recognized as DESTRUCTOR and

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

will be recognized as DECLARATOR.

TETRA uses an algorithm, by which the longest match will be preferred.
By this you can avoid limitations that the top down analysis of TETRA would have otherwise.

7.8.4 Productions: block, outer_block

The ***outer_block*** production describes the body of a function.
The code of a c++ function can be extended over many lines and contain complicated nested structures. For the aim of the current project this is irrelevant. It is important only, to find the insertion positions.
The idea is, that for every closing brace there is a corresponding opening brace. So a block can be described as:

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

Inside of the pair of braces, there is a sequence of code and sub-blocks. The only difference from the body of a function to a block is, that it's an outer block, not included in a different. The positions after it's opening and before its closing brace are the insertion points, we are looking for. Here the rules **print_at_enter** and **print_at_exit** are called, which are used, to write the desired guard functions into the output:

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

Attention:
The option for the testing of all literal tokens must be switched off, as otherwise a token which is

needed only in other productions, can be recognized here instead of a SKIP recognition.

The functions *print_at_enter* and *print_at_exit* are defined on the element page:

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

Here the variables *m_sScope* and *m_sName* are used for the construction of the guard.

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

Here the variables *m_sScope* and *m_sName* are cleared.

7.8.5 Improvement: '{' and '}' in strings

A complication arises, if there are quoted braces, e.g. "{" or "[{}]". These characters would be interpreted wrongly as opening or closing braces. The alternatives inside of a block therefore have to be extended by strings:

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

The ***block*** production and accordingly the ***outer_block*** production now look as follows:

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

By means of this improvement also the code can be parsed, which is produced from the *guard* project.

7.9 Bill

You should know the most essential operation elements of TETRA.

In this project sub-expressions of regular expressions are used. This will be possible in the future only in the standard and professional version of the TextTransformer.

Problem definition:

The sum of the positions of a bill shall be calculated. The amounts of the bill use a comma (German localization) to separate the fractional digits. There is nothing really new inside of the project, but it demonstrates a further kind of applications of the TextTransformer.

TETRA Program:

The project is at:

\TextTransformer\Examples\Bill

This simple project uses actions immediately combined with tokens, where the texts of amounts are converted into the according numbers.

7.9.1 Production

The only production of this project is constructed according the meanwhile well know scheme of a loop containing a SKIP symbol.

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

The variable of the type double is passed to the token Amount and Amount_, where the value of the respectively recognized amount will be added.
The whole sum is printed at the end of the program.

7.9.2 Tokens

Amounts can have fractional digits as in the following examples:

23,8
1,35
365,-

Amount_

For amounts, where the fractional digits are replace by a hyphen, a special token must be defined, because a hyphen cannot be converted into a number directly.

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

The token is defined as a sequence of digits followed by a comma and a hyphen. Optionally a space can precede the hyphen

The digits before the comma only determine the value of the token. Because of the parenthesis around "\d+" they can be accessed by xState.str(1). The return value of this expression is passed to "stod" immediately, which converts the string containing the digits into a double value. This is added to the sum in xSum.

Amount

Normally an amount contains fractional digits. These are recognized by the token Amount:

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

The comma can be followed either by one ore by two digits. The function can't convert expressions containing a comma (used in Germany). So temporarily the text presentation of the number is converted into a text, which can be converted by stod.

7.10 XML

You should know the most essential operation elements of TETRA.

Problem definition:

A parser for XML documents shall be build.
XML (eXtensible Markup Language) is a widespread standard for a formal language, which describes structured data and their formatting. So XML enables an exchange of these data between different applications and over the WEB.

TETRA Program:

The project is in the directory:

\TextTransformer\Examples\XML

ISO_XML.ttp is a first version of the project which is narrowly the wording of the XML standard.

XML.ttp is the revised version.

By this project the construction and evaluation of parse trees is demonstrated

7.10.1 ISO-XML

On this page a few hints will follow, how the TextTransformer XML parser derives form the standard specification of XML. Who isn't interested at these details can continue to the next page.

The **XML standard** is described in detail at

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

To specify XML an Extended Backus-Naur Form (EBNF) notation is used, which again is standardized (see: http://www.cl.cam.ac.uk/~mgk25/iso-ebnf.html).

The **standardized EBNF notation (ISO-EBNF)** is fortunately is similar to that of the TextTransformers, but isn't conceived for practical use. ISO-EBNF at first is a very elementary description, without a distinction between tokens and productions. Secondly it is not taken into consideration, whether the grammar is deterministic recognizable, especially, the grammar don't conforms to the LL(1) condition.
To transform the XML grammar three steps are necessary:

1. an import project (quick and dirty) similar to the project for the cocor import, by which the ISO-EBNF-XML rules can be imported as TextTransformer productions.
2. all productions, which only are describing character sets, are transformed to tokens (see remarks below).
3. LL(1) conflicts are solved, similar as described for the parser of email addresses

A further problem is, that an XML documents in principle supports Unicode, which the TextTransformer at the moment still doesn't. (The option to create parser code on basis of wide characters is in work.). But the first 128 characters of the ASCII-Code and of UTF-8 coded Unicode are identical. So the Tetra XML parser will read most XML documents in spite of simplified token definitions.

Some further remarks concerning the transformation of ISO-EBNF:

In ISO-EBNF there is an operator without counterpart in the syntax of Tetra:

A - B matches any string that matches A but does not match B

A translation of this operator is simple, if *A* and *B* are characters or character sets. Then *A - B* can be combined into one set of characters.
If *A* and *B* are sequences of characters, *B* can either be a permitted alternative of *A - B* or an occurrence of *B* in the input is an error.

Example:
```
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]
```

Frequently ISO-EBNF defines character sets as sequences of alternative characters. As far as possible these should be combined to a common set by '[' and ']'. This will accelerate the scanning of a text very much.

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

The character set *S* of the example just given even can be deleted from the project. *S* is just the set of ignorable characters. This as such is not specified from ISO-EBNF. Each position of the grammar, where *S* can or must occur is specified explicitly. So the grammar becomes quite confused an in addition many LL(1) conflicts arise.

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

After *VersionInfo* is recognized, there are three possibilities to continue with S. If however *S* is defined as ignorable, the rule is LL(1) conform. 'encoding' | 'standalone' | '?>'
follows directly on *VersionInfo*.
If *S* is defined as ignorable, concatenations of characters, where *S* may not be inserted, should be

combined into one token.

Example:

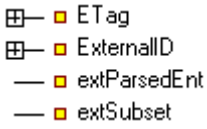
```
ISO-EBNF:          EntityRef ::= "&" Name ";"
Tetra:             EntityRef ::= &{Name};
where {Name} is a macro for the token name..
```

One problem remains. There are some spaces required at some positions of the ISO-EBNF-specification. You could define special tokens ending with a space. But the elegance won by the introduction of the ignorable characters then partially would be lost again. As the XML-parser example is not thought for verification of XML conformity, but to read and process XML documents, this point isn't really a problem.

7.10.2 XML document

The start rule for the XML parser is: ***document***. After you have parsed the start rule as usual by

not all names in the syntax tree are preceded by a square to open the structure of the production.

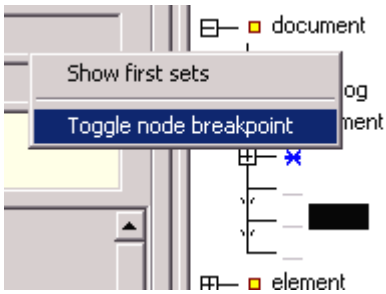


The reason is, that the XML grammar consists of two overlapping parts:

- the part for the real document and
- the part for a DTD (document type definition), which is written in an external file and defines tags and attributes for the document

Many XML documents don't need external DTD's and here only the first part shall be examined. The other rules have remained for interested user in the project.

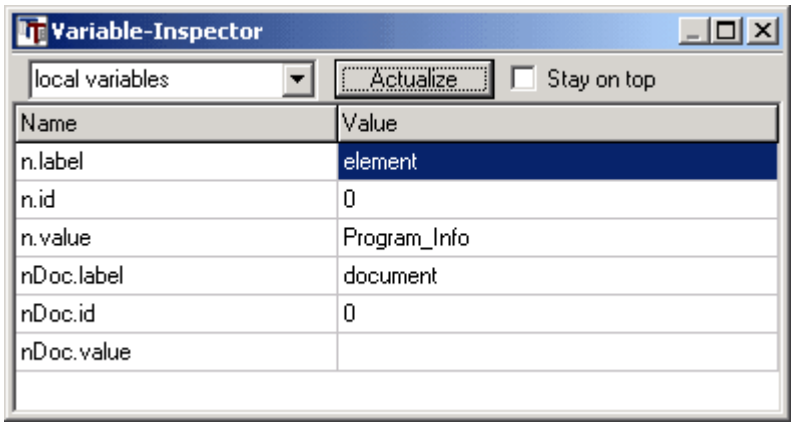
At first glance both the grammar and the document appear quite confusing.
The TextTransformer project can help, to make them clearer. The structure can be shown in the variable-inspector. For this a node breakpoint is set on an action at the end of the start rule.



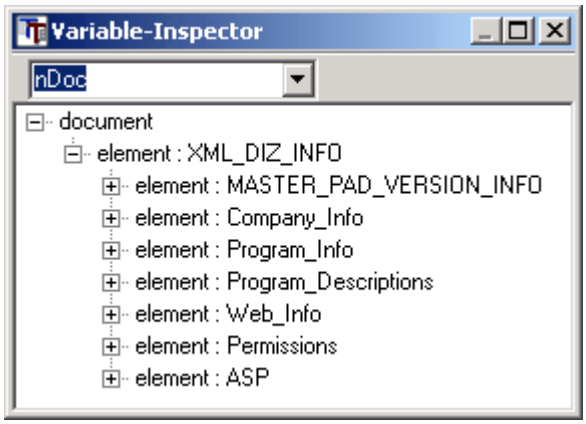
Now you can execute the program up to the breakpoint

and open the variable inspector.

Now choose to view the local variables:



and double click on the value side on one of the *nDoc*-lines.



The result is a tree view of the XML document.

7.10.3 Tree generation

nDoc, which just has been shown, is declared in the *document* production:

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

node is a structure, which is characterized by a label (here: "document") and an *str*-value. In addition nodes have the special property, that they can be combined to trees. Such combinations with other nodes happen in the productions, which are passed while the parsing is done. First *nDoc* is passed to the production *element*:

```
element[ nDoc ]
```

There another node with the label: "element" is declared:

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

which is passed to the *content* production:

```
content[nElem]
```

After the *content* production has been processed *nElem* is added to xNode (the *nDoc* passed from *document*) as a child.

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

In the same manner meanwhile child nodes are added to the node nElem itself, while the content production was processed. So the whole tree is generated, which then can be shown in the variable inspector.

An alternative inside of the *content* production is *CharData*. The token *CharData* builds a leaf of the tree. Here a node object is created too, but in contrast to the other nodes this node gets a value.

7.10.4 Tree evaluation

To transform the structure of a document into a tree at first, instead of transforming it directly into the desired form, has the advantage that an access on the data is possible now in arbitrary order and repeatedly. The output of the data can be organized now very systematically and to write the same data in different formats, e.g. Html and Rtf, can be done by similar procedures.

In this exercise example the XML document shall simply be issued in the form of simple text.

The procedures for the output of the tree data are assembled into a common container.

```
mstrfun           m_PrintText;
```

The *mstrfun* is a function table: *mstrfun* contains member functions, which will be applied on nodes, the label which of is the key of the function in the table.

The table is initialized as follows:

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

That means, for a node with the label "document" the function with the name "DocText" shall be executed; for a node with the label "element" the function with the name "ElementText" shall be executed ... In the first instruction a default function is added to the table, which shall be executed for all nodes with a label not contained as key in the table.

The functions *DocText*, *ElementText*, *CharDataText* and *DontPrint* all are defined on the pagefor class elements. The function *DocText* has to be called for the root of the tree:

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

Here for all child nodes of the root *m_PrintText.visit(xState, pos)* is called.
The *visit* method of a function table is the cardinal point of the whole tree evaluation. This method redirects the node argument pos to the function, which matches the label of the node. You can read the *visit*-function in this example as an abbreviation of:

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

The function *ElementText* is constructed as *DocText*, with the difference, that the value of the node is issued:

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

before the sub nodes are visited. So, beginning at the root node all nodes are passed systematically up to the leaf nodes. There the function *CharDataText* outputs its value.

7.10.5 Character references

Inside of an XML-element:

```
<text>  ...  </text>
```

the characters:

```
< > " ' & $
```

may not be used.So the have to be coded either as a name entity or as a decimal entity:

Character	Name entitiy	Decimal entity
<	&lt	&#60
>	&gt	&#62
&	&amp	&#38
"	&quot	&#34
'	&apos	&#39

The mstrstr class variable *m_EntityRefs* with the values of the first and second column is used to decode the named entities. Insde of the *Reference* production *m_EntityRefs* helps to translate the named entities into the corresponding characters.
The according decimal entities are treated in the action for the token:

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

Special character, which don't belong to the first 128 characters of the ASCII set, often have to be coded too.
Whether and how this is necessary depends on the encoding attribute in *XMLDecl*. A complete XML parser should be able to access a lot of tables. It is presupposed here for the demonstration, that we are using the standard font for Western Europe, Latin America (ISO 8859-1) . The characters then can be translated according to the numbering of the ANSI table.
The regular expression *CharRef* either recognizes a character in a decimal coding and delivers the corresponding decimal number as the 1. sub-expression or it recognizes a hexa decimal in the 2. sub-expression.

```
{{
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 Comments and processing instructions

There still is a number of flaws in ISO_XML.ttp which shall be removed now in the transition to XML.ttp.

- The superfluous productions and tokens are removed
- Cryptic abbreviations such as "PI" are replaced by more meaningful names: ProcInstr (= Processing Instructions)
- Comments and processing instructions are dealt with as a part of the ignored characters or as inclusions.

The last point is for didactic purposes. It would be in other grammars of a greater use than for XML, where these inclusions may happen only in places specified exactly.

Comments and processing instructions have a special role: they can be included in many places in the document without changing the data, which are transported by the XML dorcument to an apllication; they are containing additional informations.
The comments are meant for the human reader and can be ignored by the application.

The regular expression for the comments can be combined with the other ignorable characters into a common expression

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

Processing instructions contain information for external applications - e.g. complete php scripts can be embedded here - and can be put as an inclusion production.

The new expression and the inclusion production can be put in the **global project options**. (*ProcInstr* then must be removed in the local options of itself). The parser then tolerates XML documents, though, where e.g. a comment occurs inside of a tag.
If such an occurrence shall cause a fault, the productions must be changed so, that their local options can be modified so, that exactly the permissible occurrences of comments and processing instructions are parsed. Whether there are characters to exclude or whether an inclusion follows, always is checked with the determination of the next token. So the local options of a production are effective as soon as within the production a new token is looked up. Since e.g. *content* can start with *comment,* the token, which is the last token before a *comment* in the XML syntax, must be the first token of a production, which checks for comments.
Therefore the additional production *element_content* is defined and analogously the additional production *doctypedecl_core*. The local options for the following productions are adapted so that comments and processing instructions are recognized in them.

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

```
element_end ::=  "/>" | ">" element_content

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

prolog ::= XMLDecl? doctypedecl?
```

Please notize that comments and processing instructions are also recognized in and after *prolog* production since the successors of production calls are checked explicitly too.
The *element* production also is changed a little now. However, no local options are put in it.

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

7.10.7 Insert client data

A real application for the evaluation of the XML tree is the transportation of data from customer demands in an insert instruction of the database language SQL. The texts Client1.txt -- Client5.txt are examples for these demands. These texts aren't complete XML documents. After an introducing text, the customer data follow in an in an incomplete XML form.

The start rule ***Clients*** jumps by SKIP directly to the beginning of the XML part:

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

The already known *element* production is called then. The construction of a SQL instruction is finally carried out in the function: *PrintSQLInsert*. By

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

the table and the fields, which shall be filled with values, are specified. The values are then selected from the tree in the order in which they are used. A function table isn't necessary for this simple tree. E.g.:

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

So an insert instruction is got, by which the values can be inserted into a database. E.g.:

```
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" );
```

By a N:1-transformation in the transformation manager the insert instructions of all five example files can be written into a single text file, so that all data sets then can be inserted into the database at once.

7.11 Unit_dependence

You should know the most essential operation elements of TETRA.

Problem definition:

Texts in different files frequently depend on each other. Programming languages are a typical example of it. Here the dependence is indicated by so-called Include directives. In this project shall be demonstrated, how to use these directives to access the presupposed texts. A list of all the files a Pascal source text directly and indirectly depends on shall be produced. To get this list, it is necessary also to parse all presupposed texts.

TETRA Program:

The project is in the directory:

\TextTransformer\Examples\Unit_dependence

In this project some of the commands are used for the path and file treatment and it is shown how texts are loaded to process them in sub-parsers.

7.11.1 Productions

In the programming language *Pascal* the presupposed Pascal-Units are listed behind the keyword *uses*. E.g.:

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

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

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

unit_name ::=
   IDENT
```

The code is skipped with *SKIP* until the keyword *uses* is found and then the list of the included Units is parsed and until the end the text is skipped again.

7.11.2 Containers and parameters

At first the program must be told, in which directories the units have to be to looked up. So a Vector container is defined on the page for the class elements

```
vstr  m_vIncludeDirs
```

The user has to add the according directories to this vector, before he starts the program. E.G.

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

The found files must be noticed someplace. So a mstrstr container is defined:

```
mstrstr  m_mUnitPaths
```

To every name of a unit as a key the accompanying path as value can be stored in this map. Then later can be found out easily whether a unit already was searched, when it appears again in another unit.

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

The case that the path isn't found also must be noticed to not look for it once more. Therefore there is a second map:

```
mstrnode  m_mNotFoundUnits
```

Mstrnode is chosen here as a container type. Two strings which could be of interest can be stored to the node value at once: the unit, in which the not found unit was listed and a level parameter, which tells how many files must be opened to come to the unit from the original source file.

To attain these two information, the productions mentioned above must be provided with corresponding parameters.

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

The first unit production is then called with the current values in the start rule:

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

7.11.3 Include files

Everything is prepared for the decisive step now. If the name of a unit was found and it wasn't searched for the corresponding file yet, by the function find_file the unit is lloked up in the include directories.

```
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))
  {
  ...
```

With change_extension the pas-extension is appended at the name, as for Pascal files usual. As a precaution at first with is_directory is checked whether the directory is available on your computer. If you haven't modified the directory list (see above) or cannot because no Pascal is installed presumably, then the program is stopped in this place.

If the search was successful, then the reference variable *sPath* contains the wanted path. By means of load_file the file is loaded into the string *buf* now. It is the whole point now that the unit production like a normal function can be called to parse the new text. It serves as an "sub-parser" with that:

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

While the unit production has only two parameters in the main parser, it still gets the additional text parameter as sub-parser for the call in first place. In the sub-parser for further included units now is searched just the same as before in the main parser.

7.12 Java

You should know the most essential operation elements of TETRA.

Problem definition:

The programming language Java 1.4 shall be parsed.

TETRA Program:

The project is an adaptation of a Coco/R project:

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

The TextTransformer *Java.ttp* project is in the directory:

\TextTransformer\Examples\Java

In this project look-ahead productions are used and some special variants of looking ahead are explained.
The use of the tree wizard and of the function table wizard is demonstrated step by step. First the code for the creation of a parse tree is generated and then a simple transformation program (copying program) will be made.

7.12.1 Coco/R adaptation

The Coco/R adaptation was carried out automatically by means of a TextTransformer program, similarly as described for the older Coco/R version. The IF constructs of Coco/R, however, aren't equivalent to those of the TextTransformer and because of their rare use the overhead for the development of an automatic translation for them isn't worthwhile. Because of this defect the transformation program isn't at the disposal for download. On enquiry, however, you can get it free of charge.

7.12.2 Simple look-ahead production

The Java parser isn't LL (1) conform. The decision on the alternative to be chosen depends on a look-ahead of more than a single token. The IF and the WHILE construct of the TextTransformer allows such a foresight if in the respective condition a production is invoked for the look-ahead.

For example, if an identifier is recognized as next token in the *statement* production, then it isn't clear at first whether this identifier represents a label or an expression. It represents a label if a colon follows it.
So the progress is made dependent on the production *isLabel*

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

isLabel can parse the following text exactly, if a colon follows the identifier. Altogether, the grammar alternatives are therefore tied into the following IF construct:

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

7.12.3 Negative look ahead

In the production *ArrayInitializer* a WHILE loop is called:

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

```
END
```

where

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

This means, the loop is executed as long, as a comma is following but no closing curly bracket follows the comma. If no comma follows, then *commaAndNoRBrace* cannot parse the current text. If a comma is following and a closing curly bracket '}' follows the comma, the look-ahead production *commaAndNoRBrace* also returns false. In this case this is forced by *EXIT*.

7.12.4 Complex look ahead

In the block statement production the same production is called for a look-ahead, which shall be executed in the success case:

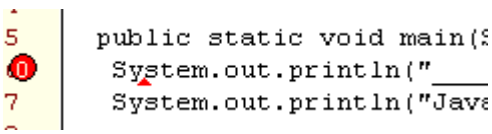
```
IF( LocalVariableDeclaration() )
  LocalVariableDeclaration ";"
ELSE
(
   ClassOrInterfaceDeclaration
  | Statement
)
END
```

Attention: depending on the fact whether the testing of all literal tokens is activated in the project options or not, the call of *LocalVariableDeclaration* for a look-ahead can have different results. For example, if the token *return* - one of the alternatives of *Statement* - follows, it will be recognized as an identifier *ident*, if not all literal tokens are tested, but only the first set of *LocalVariableDeclaration*. So e.g. "return true;" would be interpreted as a local variable declaration, what surely is wrong.

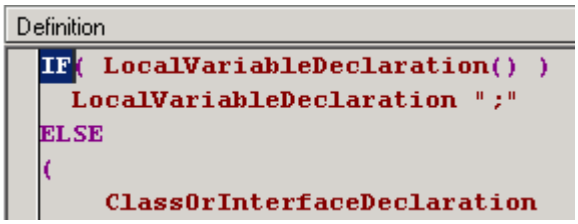
Remark:
Indirectly further look-ahead parsers can be invoked while testing, whether LocalVariableDeclaration matches the actual text or not.

7.12.5 Debugging a look-ahead

You can see what happens in detail when a look-ahead is tested, if you step into the look-ahead with the debugger. First you can put a breakpoint on the word "System" in the sixth line of the example text.

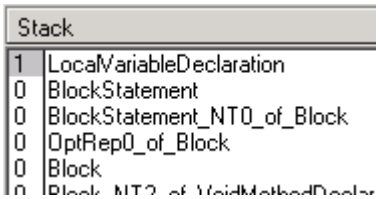


The project then can be run up to this point. After another three single steps the "IF" is marked in the production "BlockStatement".

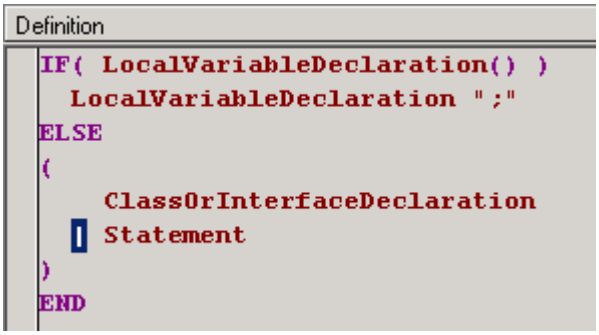


Now can step into the look-ahead. After a click on the corresponding button the debugger changes to the production "LocalVariableDeclaration" and a '1' is shown in the little field right next to the button.

This means that the debugger is in the first level of a look-ahead now. This is indicated by a '1' with a gray background in front of the name of the production at the top of the stack in the window for the stack too.



The debugger can be executed just the same as used for the main parser now within the look-ahead. Behind the word "println", however, the parser cannot continue: no following token is recognized. The look-ahead has failed and is left. The debugger again is in the production "BlockStatement" now but in the ELSE-branch of the IF-structure.



The field for the indication of the level of the look-ahead is empty, i.e. the debugger shows the progress in the main parser again.

7.12.6 Parse-Tree

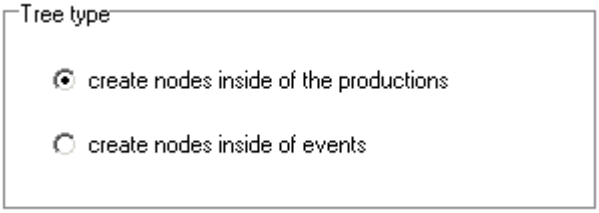
The Java parser is a quite large project and a lot of work can be expected, to make a complete transformation program from it. The wizards of the TextTransformer can help greatly here.
As an example, at first the code for the generation of a parse tree shall be inserted in the project by the tree wizard. With the function table wizard then functions can be created by which the parse tree can be evaluated.

It is frequently recommendable to create a transformation program at first that simply copies the source files. With such a program the parser and the parse tree can be tested well: the program works correctly if the transformed files are identical with the source files. At the copy program then simple modifications can be made, the results of which can also be verified easily.

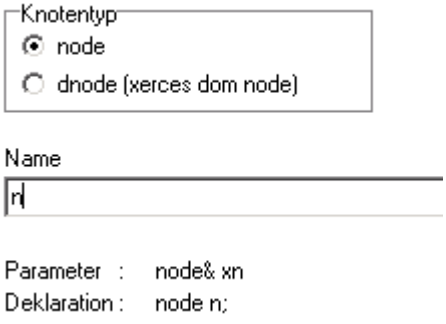
Who don't like to reproduce the following steps in detail can load the ready result made by the tree wizard also directly:

...\TextTransformer\Examples\Java\JavaTree.ttp

If you call the tree wizard in the HELP menu, then a choice appears for the manner, how the tree shall be created. You can leave this option.

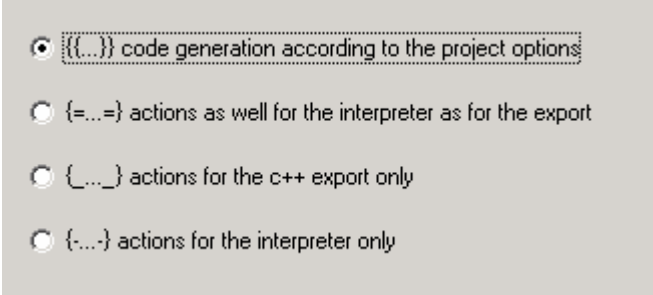


Next a choice appears for the node type and an entry field for the node name at first. *node* is elected as type and *n* is selected as name here:



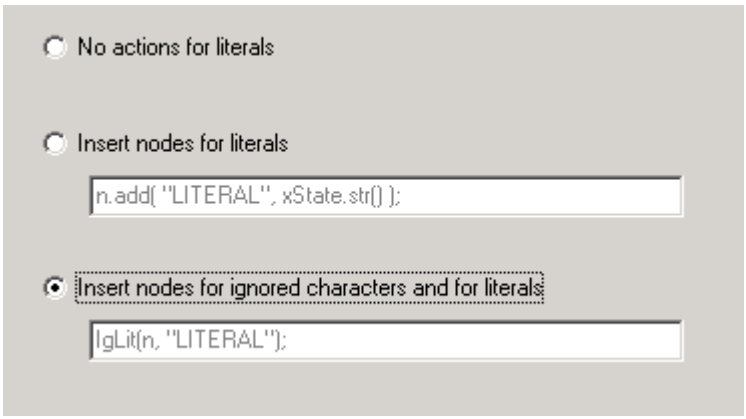
Below of the name field you can already see, how the node parameter in the parameter field and the node declaration in the text of the production will look like.

You leave the presetting on the next page of the wizard:



Then choose the complete option on the next page:

Three options can be chosen for the treatment of the literal tokens on the next page of the assistant.



For the copy program the undermost point must be selected. The semantic action *IgLit* which takes care that both a node for the ignored characters and a node for the recognized text are added to the tree, is then inserted after every occurrence of a literal token in the project.

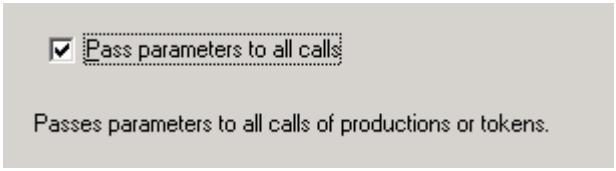
The *IgLit* function then will be inserted on the element page..It looks like:

```
Name: IgLit  // Ignorierter Text und Literal
```

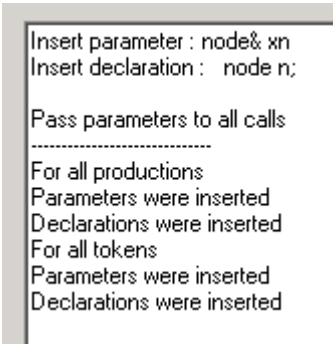
```
Parameter: node& xnNode, const str& xs
Text:
{{
  node n("IgLit");
  xnNode.addChildLast(n);
  n.add("IGNORED", xState.str(-1));
  n.add(xs, xState.str());
}}
```

A sub-node with the label "Iglit" is added to the tree node xnNode and this sub-node gets the nodes for the ignored text and the text recognized by the token.

On the next page activate the check box: Pass parameters to all calls.

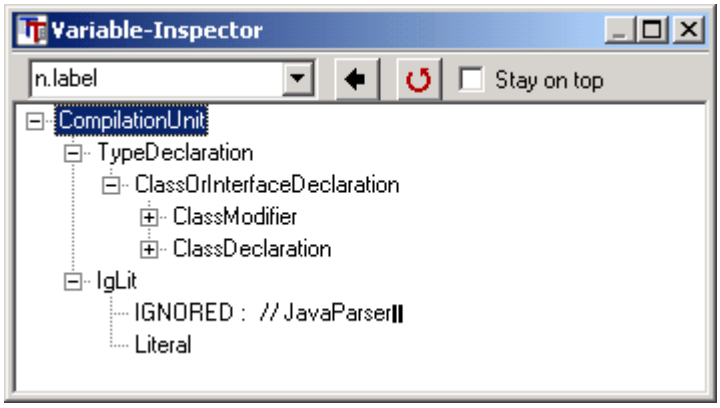


Then you can go to the last page and click on the *Finish* button.



After the results are shown you can close the *Tree wizard* by the *Cancel* button.

You can examine a generated tree after execution of the program with the start rule *CompilationUnit*, as demonstrated for the XML example in the variable inspector:



Remark: At the end of the start rule *CompilationUnit* explicitely the symbol **EOF** is set. So the wizard insertes an action for this symbol too:

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

In this last action the ignored text will be inserted in the tree, which follows on the last symbol of the Java grammar:

the comment: `"// JavaParser"`

7.12.7 Function-Table

By the function table wizard you now can insert a frame for evaluating the parse tree.

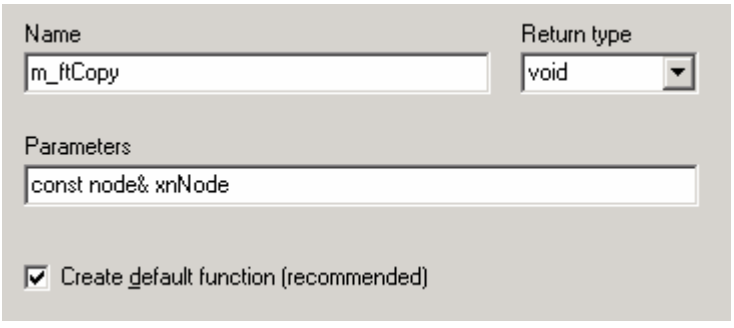
Who don't like to reproduce the following steps in detail can load the ready result made by the function table wizard also directly:

...\TextTransformer\Examples\Java\JavaCopy.ttp

If you study the parse tree in the variable inspector, you will recognize, that it consists of branches with the names of productions and there is a node for every token with the label: *IgLit*. A function table shall be produced, with functions for the treatment of the nodes of all labels.
The *IgLit* nodes are primarily important for the copy program since the complete source text is available in the two sub-nodes. All the production nodes can be treated uniform: they serve as inter-stations for the iteration to the IgLit nodes.
So only two functions are needed, altogether:

Default function for the treatment of the production nodes
***IgLit* function** for the treatment of the IgLit nodes

On the **Start page** of the function table wizard select: Create new table. So you come to the second page. The fields of this page should be filled as depicted:

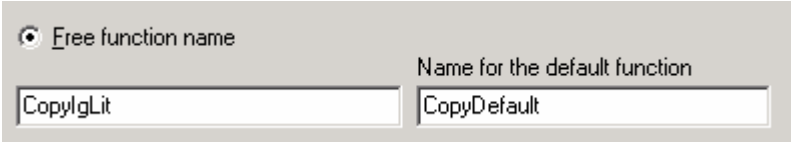


On the next page select : Function for a single label

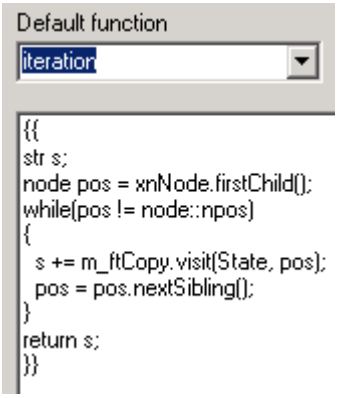
On the next page write the label: **IgLit**

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

On the next page write the function name: *Copy*IgLit and for the name of the default function: *CopyDefault*.



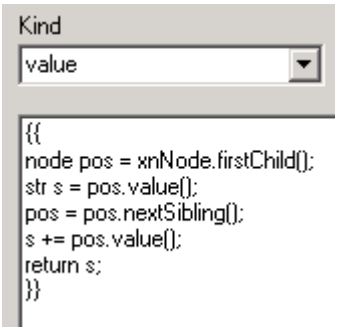
As default function choose *iteration* on the next page.



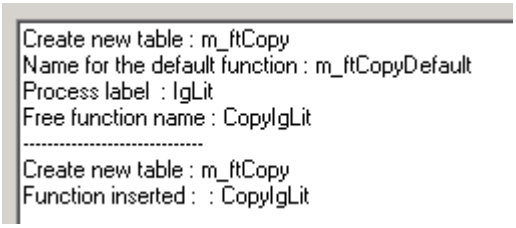
This function is for the iteration to the *IgLit*-nodes.
On the next page choose *value* and change the text to:

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

This function writes both parts of text of an *IgLit*-node.



Now you will come to the last page and click on the *Finish* button.



By *Cancel* you can close the function table wizard now.
On the element page you can see, that a function table and two functions are inserted.

The copy project nearly is ready now. Unfortunately, there is a flaw. On the token page the tree wizard had inserted code for the addition of the token strings only and nor for the ignored characters. So you have to change the actions on the token page manually to:

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

The last thing you have to do is to call the evaluation of the tree. Change

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

in the action at the end of *CompilationUnit* to:

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

If you execute this project the java source code will be copied into the output window.

7.13 C-typedef

You should know the most essential operation elements of TETRA.

Problem definition:

New names can be defined as an abbreviation of more complex expressions within a text sometimes. Such names shall be inserted during parsing as additional tokens.
Type definitions in programming languages are a typical example of such abbreviations. In this TETRA project this is demonstrated for the language C. For better clarity the rules of C were reduced verry much The complete C grammar is available soon at

http://www.texttransformer.org

TETRA Program:

The project is in the directory:

\TextTransformer\Examples\Typedef

The use of dynamic scanners with placeholder tokens and text scopes is demonstrated in this example.

7.13.1 Typedef

A type definition is started in C with the keyword *typedef. For example:*

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

cpchar can be used instead of "const char*" in the C code after this definition.
The simplified production *type_definition* for parsing the definition is:

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

The second line contains the rule for the expression, which has to be abbreviated. *ID* in the third line recognizes the name of the definition.
With the following semantic action the found name is added to the dynamic token "TYPE" as an additional alternative now. *TYPE* is defined on the token page as:

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

TYPE is used in the *type_specifier* production:

```
   "void"
|  "char"
|  "short"
```

```
| "int"
| "long"
| "float"
| "double"
| "signed"
| "unsigned"
| TYPE
```

So:

```
cpchar p;
```

p is correctly recognized as a declaration of a variable with the user defined type *cpchar*.

7.13.2 Scopes

The third parameter in the call:

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

wasn't explained yet. The area within which the additional token is recognized can be limited with this parameter. It is an optional parameter. If this optional parameter is left out, the definition applies to all following code. If the third parameter is passed, however, the definition is then valid only for time time as an area is defined with the corresponding scope name. Such a scope is defined at the very beginning of the start rule *translation_unit* and it is removed at the end:

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

Likewise happens in the compound_statement: These scopes are numbered, to be able to generate unique names for them. Scope names are managed with a stack. The command *PushScope* puts an additional scope on the already available stack of scopes and *PopScope* removes the topmost scope. A dynamic tokens is recognized as long, as the scope, for which it has been defined is still in the stack.
The type *pchar* is defined in the following example within the first *compound_statement*.

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

This definition isn't, however, valid in the second *compound_statement* any more so that its use leads to a fault. This is exactly the mechanism of type definitions in C.

7.14 TETRA productions

You should know the most essential operation elements of TETRA.

In the directory

"\TextTransformer\Examples\Productions"

is the project for the TETRA script language. All actions are removed.

7.15 TETRA-EditProds

You should know the most essential operation elements of TETRA.

The project in the directory

"\TextTransformer\Examples\EditProds"

is an example for a parser, which can be used for different purposes, by producing at first a parse tree. The project EditProds can edit TETRA productions in two manners, which are nearly opposed to each other:

1. insert tree nodes into productions
2. delete semantic actions form productions

7.16 TETRA interpreter

You should know the most essential operation elements of TETRA.

In the directory

"**\TextTransformer\Examples\Interpreter"**

is the project for the TETRA interpreter. All actions are removed.

The parser for the interpreter is based on a grammar, which you can find at:

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

You should know the most essential operation elements of TETRA.

In the directory

"\TextTransformer\Examples\ImExport"

is the project for the TETRA Import of ASCII text files, which were previously exported from the TextTransformer. All actions are removed.

The format for the exported files is provisional and probably will be replaced by a XML format.

7.18 TETRA-Management

In the directory

"\TextTransformer\Examples\Management"

is the project, which is used to parse a management.

7.19 Cocor import

You should know the most essential operation elements of TETRA.

This example is for advanced users of the TextTransformer and the compiler compiler Coco.

The TextTransformer was inspired by the compiler compiler Coco/R and is related to it in many respects. The productions of a Coco/R compiler description can be translated into the syntax of the TextTransformer essentially without problems.
This is the task of the project in the directory:

"\TextTransformer\Examples\CC2TT_17"

An example source is the script **Cr_17.atg**
(originally: Cr.atg. The 17 is the version number of Coco/R. The Java project is an adaption of a newer Coco/R project.) In this script the syntax of the Coco/R compiler description language itself is defined.

To parse the script, the TextTransformer has to adapt definitions of the script:

the same ignorable characters must be defined
the same tokens must be recognized
the productions must be parsed in the same manner

7.19.1 Ignorable characters

In Coco/R the ignorable characters and comments are defined separately. (The space character is always ignored.)Additional there are pragmas to control the compiler, which may occur at arbitrary positions inside of the source. In the script Cr_17.atg this is written in the lines:

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

In the TextTransformer the ignorable characters, comments and pragmas are combined to one expression and set in the project options:

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

results in a new IGNORE:

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

Remark: Nested comments cannot be defined in the TextTransformer.

7.19.2 Tokens

An automated translation of the token specification of Coco/R into the regular expressions of the TextTransformers in principle should be feasible.
But this is not, what shall be done here, because this would mean a considerable effort, in particular, because of the different manner in which character sets are defined. Furthermore the definitions of token are only a little part of a translating project.
The token definitions of the Coco/R compiler description shall be translated here directly.

In Coco/R at first the used character sets are defined and then used by an EBNF-definition of the token. In the TextTransformer this two step procedure could be applied too, but normally the token are defined together with its character sets. Hereby several predefined character sets can be used, which don't exist in Coco/R.

The according lines from Cr_17.atg are:

```
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} .
```

In the TextTransformer these character sets can be expressed as follows:

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

Thus the tokens are:

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

7.19.3 Productions

Coco/R uses the *EBNF* syntax and some special symbols to define productions. *EBNF* means "Enhanced Backus-Naur-Form". This syntax in essence accomplishes the same as the regular expressions of the TextTransformer. Instead for example of the bracketing of a repeat "(...)*" a pair of braces "{...}" is used. Analog other bracketing has to be exchanged:

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

The ANY symbol of Coco/R can be replaced provisional by the SKIP symbol. But the correctness of this replacement can't be guaranteed automatically. It must be checked for each single case.
For the WEAK and the SYNC symbol of Coco/R, there is no corresponding symbol in the TextTransformer. They can be omitted, because they don't have a meaning for the parsing directly, but only for the treatment of parser errors.

7.19.4 Post processing

If a transformation of a Coco/R scripts is performed, the result can be saved as a file with the extension "ttr". Such a file can be imported into the TextTransformer.
The token have to be translated directly - as just explained - and added to the project on the token page manually.
In the automatic transformation, there might be some incorrect translations.

1.: You have to check, if all empty alternatives are followed by a semantic action.

2.: The simple replacement of the ANY symbols by the SKIP symbol might be incorrect. The according positions have to be checked.

In the Cr_17.atg project following corrections have to be done:

- (ANY)* can be replaced simply by SKIP
- { ANY | badstring } must be replaced by (SKIP | string | badstring)*. Otherwise a text including a

quotation would be taken into a piece before the closing quotation mark and a piece after the closing quotation mark, that means into: SKIP badstring.

7.19.5 Semantic actions

The semantic actions are included into the brackets {_ and _} as not interpretable code in the transformed text, because it has not to be assumed, that these actions are interpretable.
You can remove the actions totally. To do so, you have to disable the code inside of double braces "{{...}}" for the interpreter in the productions *Attribs* and *SemText*. This can be done in the **local options** of these productions. Then the interpreter will not write the code for the semantic actions into the target text. Only the code for the pure parser is generated.

Remark:

For the production *SemText* the local options are already activated, as the testing of all literal tokens has to be turned off. This is necessary as in principle *SemText* is:

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

If for example the literal token "(" would follow the opening bracket "(." in the text, it would be recognized as next token, because literal tokens have a proference before the tokens of a SKIP recognition. If however, the scanner only tests the tokens, which may actually occur according to the actual grammar rule, all other tokens of the project are no problem for the recognition of *SemText* .

Part VIII

8 How to ...

Notes to special advanced topics and uncommon concepts are made here.

Load data
Structure data
Write into additional target files

8.1 Load data

External data are needed in a project now and then. Single parameters can be submitted as a start parameter. It is, however, also possible to read larger amounts of data from an external file.

In the following example a list of surnames shall be used to decide whether a name recognized with a token *NAME* is a first name or not. This can be done with the following code if the first names are stored as keys in the map m_mFirstNames.

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

To feed the first names into the map, at first a file is loaded with the names into a string *buf* which then is parsed with the production *ReadFirstNames*.

```
{{
str buf;

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

ReadFirstNames(buf);
}}
```

The production *ReadFirstNames* is called as sub-parser within the semantic action here. The list could look like:

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

It is very simple to parse it:

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

```
  EOL
)*
```

8.2 Structure data

Neither one's own classes nor structures can be defined in the TextTransformer interpreter. If this is really required, you have to do it in the external code or in the part of code, which is exportable only. For most purposes, however, node/dnode is completely sufficient as a structure substitute. E.g. the typical data of an employee existing as strings can be stored in a node like that:.

```
node n(sFirstName, sLastName);
n.setAttrib("street", sStreet);
n.setAttrib("address", sAddress);
n.setAttrib("bithday", sBirthday);
n.setAttrib("salary", sSalary);
```

Several of such data sets can be managed as a sub-nodes in a tree and e.g. several of nodes or trees can be collected in a *mstrnode* class element too.

8.3 Write into additional target files

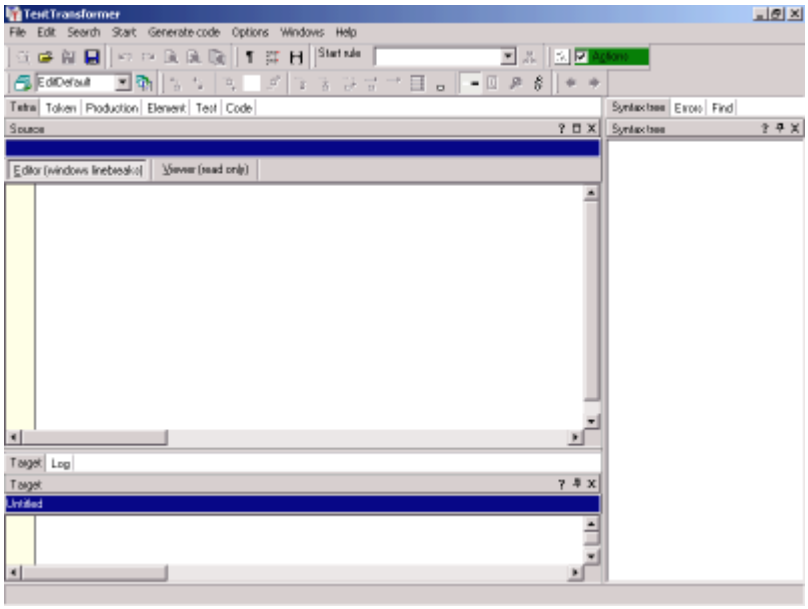
In principle, the TextTransformer is orientated at the model, that a source file is transformed into a destination file. No more files can be opened for writing at the same time in the TextTransformer interpreter. If you have to write more than one target file - e.g. a second file with logging information - you have to redirect the output and finally to reset it to the original file.

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

Part IX

9 User interface

After the start of the TextTransformer following screen appears:



The arrangement of the windows is determined by the layout "EditDefault". You can easily customize the layout to your own screen and your own needs with the mouse.

Below of the main menu and the tool bar, there are two blocks of windows, each with several tabs.

The left block contains tabbed windows for different tasks, for example the creation of TETRA rules (productions) and their execution. The **source text** to be processed is loaded into the big window on the active tab page. Later, the result of a transformation of this text appears in the **target window** lying beneath.

The right block consists in some windows for the navigation between the TETRA rules.

9.1 Tool bar

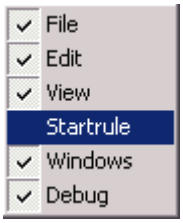
No matter, which window is active, the menu and the tool bar remain reachable. But the content of these components can change. Menu items can be added, removed or de/activated.

Some actions triggered by the menu are of general nature, but the others refer to the actual windows. For example, to save a project is a general action, but to save a text refers to the activated

editor.

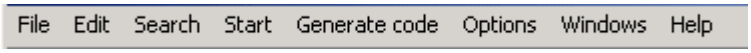
You can move the groups of buttons with the mouse and you even can undock and close such a group.

A closed group can be shown again, if you right click on the toolbar and select the according group.



The positions of the groups are saved together with the docking window layout.

9.2 Main menu

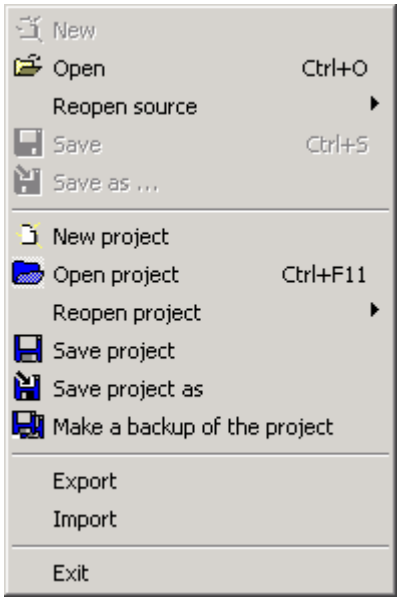


The main menu items are:

File
Edit
Search
Project
Start
Code generation
Options
Windows
Help

9.2.1 Menu: File

In the file menu texts and projects can be opened and saved.



The upper half of the menu items comprises the actions, which concern texts, and the lower half concerns the whole project.

Items, concerning texts, always refer to the current editor field! For example, if the source text window were active (the blinking cursor is in this window), the text would be opened into this window or saved from it. To save a text from the output window, this window must be active. There are other editors in the script repositories and on the editor page.

New

The current text will be deleted.

Open...

If a source file is opened, it is shown in the viewer. In other cases the text is loaded into the current editor window.

UNIX line breaks are completed automatically to Windows line breaks.

An unusual feature is the selection box ***Encoding***:

Here are three possibilities:

ANSI: the opened text is interpreted as a simple Windows text, i.e. as ANSI coded. Every byte of the file is represented in the editor as a single character of the ANSI font.

UTF-8 (ANSI): the opened text is interpreted as UTF-8 coded. TETRA cannot really process all

UTF8 encoded texts but only such that can be transformed into an ANSI coded text. Even in this mode characters are shown wrongly in the editor, if they don't belong to the ANSI set.

auto: an XML file is interpreted as UTF-8 coded, if UTF-8 inside is indicated as a coding. All other texts are interpreted as a simple Windows text.

Reopen

A text, which already was opened by the TextTransformer, can be loaded again into the current window. Hereby the text is always opened in ANSI mode (see above).

Save

To save the text of the current editor field.

Save as...

To save the text of the current editor field with a new name.
You can choose different encodings as when opening a text (see above).

New project

You can create a new project either from scratch or you can start with the wizard for the creation of a new project.

Open project

Opens an existing project.

Reopen project

To open a project, that already had been opened in the TextTransformer. A copy of the current version of the project is made automatically, with the same name and the additional extension "bak".

Save project

To save the current project.

Save project as ...

To save the current project with a new name.

Make a backup of the project

A sub-directory will be created in the directory of the project and the actual project file will be copied into it. If the project uses own frame files, they will be copied too. The names of the sub-directories will be composed of the word "Backup" and a number of three digits. The number of a new directory

will be greater by one than the greatest number in the names of the other backup directories.

Import/Export

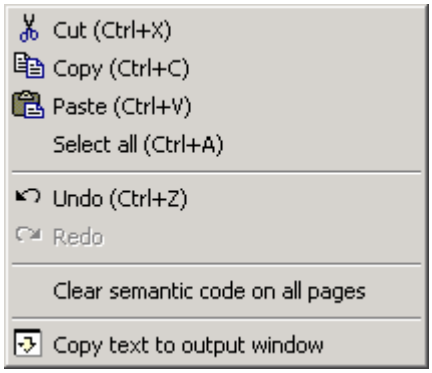
TextTransformer projects are usually written and read again with the functions mentioned above in a binary format. With the functions for the export or import, however, projects also can be written and read in a text form.

Exit

To exit the TextTransformer.

9.2.2 Menu: Edit

In the edit menu the usual items to cut, copy etc. of text are listed.



The actions always refer to the current editor field!

Undo and Redo

The functions: *Undo* and *Redo* are working in two manners:

- If the script is in the editing mode, the individual editing actions are undone or done again.
- If the script isn't in the editing mode the complete state of a script is restored which it had before changes were accepted. This is possible within one session only.

By the menu item

Clear semantic code on all pages

you can remove all actions assigned to tokens and all semantic code in the productions at once. All class elements are removed too.

By means of the item

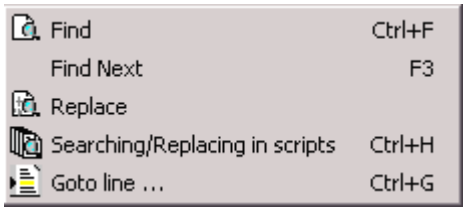
Copy text to output window

or the button

sections of text can be copied to the output window without transformation. At first the section of text must be marked and then the text is **appended** to the output.

9.2.3 Menu: Search

In the search menu are the usual items for finding and replacing text.



The actions
Find
Find Next
Replace
Goto line ...
always refer to the current editor field!

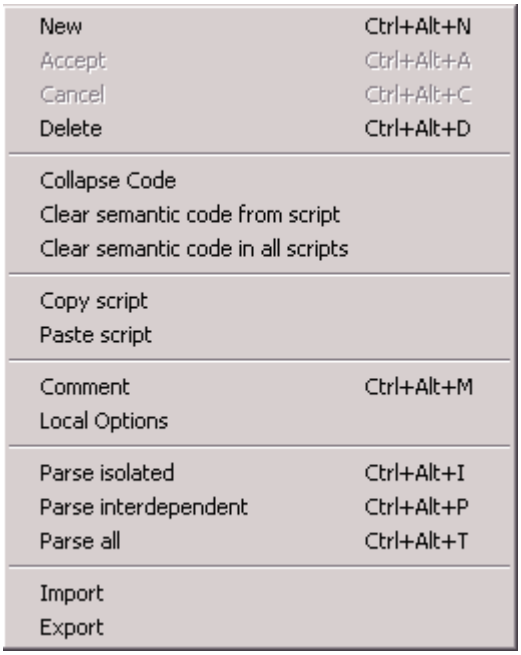
Searching/Replacing in scripts

The kinds of scripts, in which expressions shall be looked up or replaced, can be set at the bottom of the dialog. If several script types are selected, the search is carried out forward in the sequence: tokens, productions, elements and tests. In all fields of the scripts is searched. If you don't begin with the *new* button, the search starts in the current script at the current position and is then continued in the scripts with the alphabetically following names.

If a list of the search results shall be prepared, it is then published in a special window: Search. You can navigate to the corresponding position by selecting the entries there.

9.2.4 Menu: Project

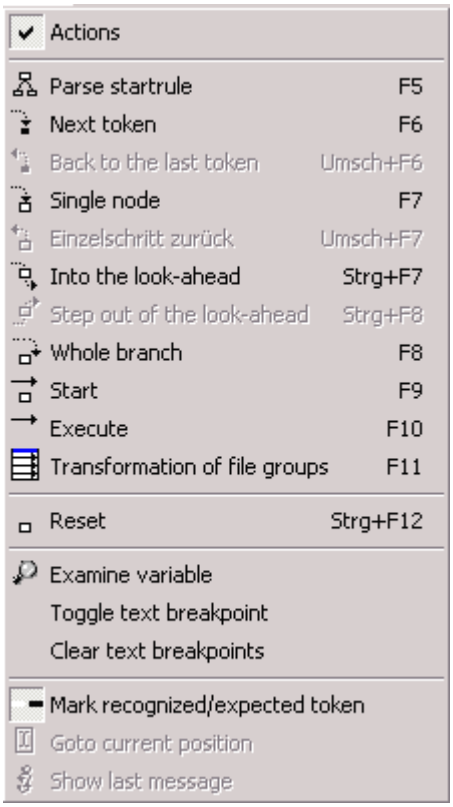
The menu *Project* only is shown if the input mask for a script is visible too. The functions for the management and compiling of scripts are summarized here



New
Accept
Cancel
Delete
Collapse semantic code
Clear semantic code from script
Clear semantic code in all scripts
Copy script . to copy a script into the clipboard
Paste script : to insert a script from the clipboard
Comment . for each script a comment can be entered
Local options
Parse isolated
Parse interdependent
Parse all
Import
Export

9.2.5 Menu: Start

The functions for debugging and executing projects are summarized in the menu *Start*

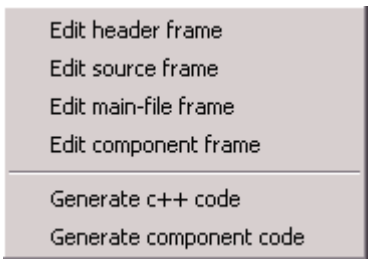


Actions
Parse start rule

Next token
Back to the last token
Single node
Single step back
Into the look-ahead
Out of the look-ahead
Whole branch
Start
Execute
Transformation of file groups
Reset
Examine variable
Toggle text breakpoint
Clear text breakpoints
Mark recognized/expected token
Goto current position
Show last message

9.2.6 Menu: Code generation

The menu contains functions for the support of the Delphi developers, which are using the TetraComponents and functions for the manipulation of the code frames and for the generation of c++ code with the professional version of the TextTransformer.



C++ code generation

Edit header frame opens an editor with the frame for the header of the c++ parser class

Edit source opens an editor with the frame for the implementation of the c++ parser class

Edit main-file frame opens an editor with the frame for a main file or another file, where the parser is called.

Generate c++ code starts the generation of the c++ code for a parser class.

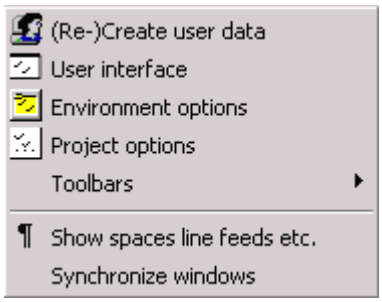
Delphi support

Edit component frame opens an editor with the frame for the Delphi support.

Generate component code creates a Pascal unit, which can be included into a Delphi application, which uses the TetraComponents

9.2.7 Menu: Options

(Re-)Creating user data



The menu Options consists of three groups of options

1. Options of the user interface
2. Environment options
3. Project options

There also are local options for single productions.

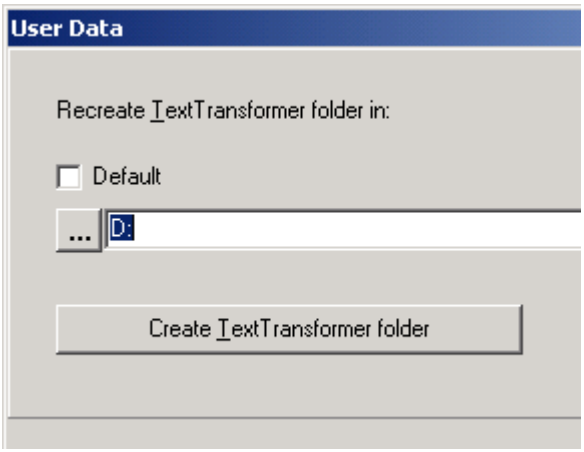
9.2.7.1 User data

There are some settings and data which can be modified by individual users without changing the corresponding data for other users. E.g. the layouts, the environment options but also the project examples are part of these data.

At the installation of the TextTransformer the original user data are written into the program folder of the TextTransformer:

```
C:\Program    files\TextTransformer\data\TextTransformer
```

When TextTransformer is started for the first time the dialog for the creation of the user data is called automatically to prepare a copy of the data in a folder that can be chosen freely by the user, as far as it is accessible to the user. It is possible to restore the folder later or to create it newly in another place.



You can start the copying of the data with the button *Create TextTransformer folder*. In the case of success the *Ok* button is activated and the *Cancel* button is deactivated.

An existing Settings folder from a previous installation isn't overwritten. The layouts of the last installation can explicitly imported into the user directory by the menu.

If the folder for the TextTransformer data has been created successfully, it has the following structure:

```
TextTransformer
              \Backup
              \Examples
              \Frames
              \Log
              \Projects
              \Settings
              \Target
```

9.2.7.2 Options of the user interface

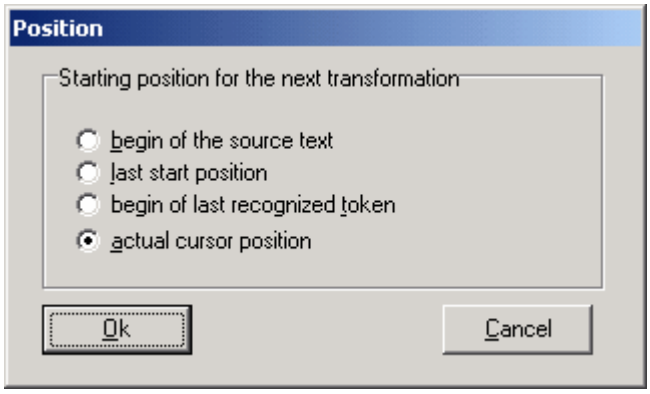
The settings of the user interface are concerning

- the way of transformations and
- the editing of projects
- the view in the debugger
- the layouts in the edit mode and debug mode

1. STARTPOSITION:

If a new transformation is started, it will begin either principally at the beginning of the input text or, if necessary a dialog is shown, where you can choose a start position.

If the source text just was opened, the text cursor is in the upper left edge of the input window. At first this is therefore the only position at which a transformation can start. As soon as the cursor has moved, might be by the keyboard or the mouse or might be by an interrupted transformation, there are several possibilities to start a new transformation. One of these possibilities can be chosen in the following box:



Only the possible options are activated, the others are grayed. If only the first option is possible, the box will not be displayed.

Transform the **whole text**
Transform the **marked section of text**
Transform the text beginning at the **last start position** again
Transform the text beginning at the position of the **last recognized token**
Transform the text beginning at the **actual cursor position**

Transforming the whole text

If you have just loaded the text from the hard disk, the cursor is at the beginning of the text and when you start the transformation, the whole text will be processed.

Transforming the marked section of text

If a section of the text is marked, when you start the transformation, only this section will be processed.

Transforming the text beginning at the last start position again

If the cursor is not at the beginning of the text, when you start the transformation you can chose to

begin again at the position, where you started the last time. This is useful for the repeated testing of a certain production at a certain part of text.

Transforming the text beginning at the position of the last recognized token

This is another option of the dialog above, which allows the interactive transformation of a text. The transformation can be done section by section. After you have processed one section, you must reset and choose a new start rule. Now you can continue at the last token.

Transforming the text beginning at the actual cursor position

This option of the dialog above is also useful for an interactive transformation. After an interrupt of the transformation and a reset, you can choose a new start rule and continue at the position of the mouse cursor. If you have not moved the mouse after the stop, the transformation will be continued at the end of the last recognized token.

2. OUTPUT:

After a transformation was executed, the result of the transformation is in the output window. If you immediately push the reset button after the transformation, then a dialog box appears, which offers deleting of the output text.
If the output window isn't empty at the beginning of a new transformation, then there are three options for TETRA to behave:

1. Deleting output text without demand
2. Appending the new output text at the old ones
3. Displaying a selection dialog

9.2.7.2.2 Editing

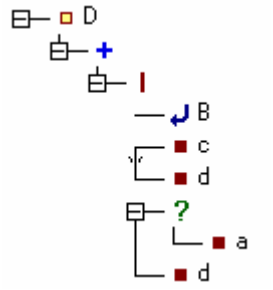
Accept changes in scripts automatically

If a script is in the editing mode, i.e., if it is new or was just changed, then the changes must be accepted before other actions can be executed, e.g. before another script is changed to or before the project is compiled. If the option *Accept changes in scripts automatically* is activated, the changes of a script are accepted automatically.

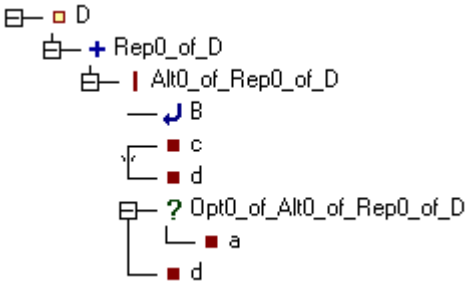
9.2.7.2.3 View

1. SYNTAX TREE:

Here you can determine, if - after compilation of a production - the names of all nodes are displayed in the syntax tree or, if only the names of the terminal, SKIP and non-terminal nodes (respectively their branches) are displayed. In the last case the syntactical structure is to be seen clearer. The names of repeats and alternatives only are important if you want to see the relationship to produced c++ code.



non/terminal names only



all names

2. MARKED TOKEN:

If a transformation is executed step by step, the current position is marked by a token in the input text. Depending on setting this is

the token recognized last, or
the next expected token

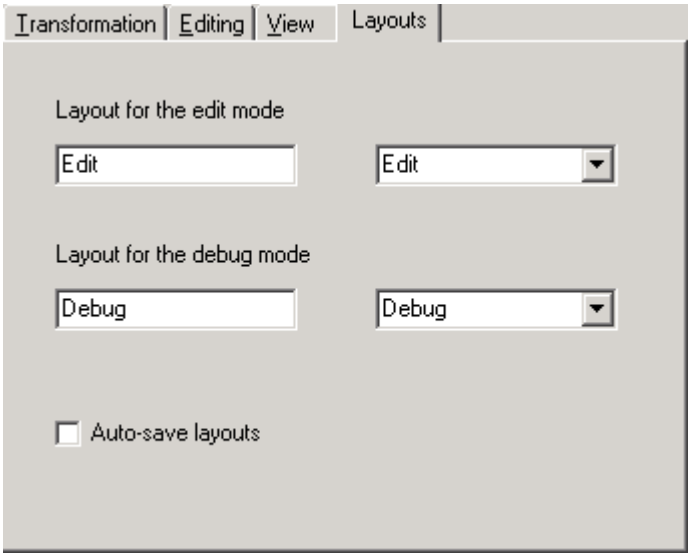
A default setting, which is active at the start of the TextTransformer can be set here. However, this value can always be changed during debugging by the button:

 show recognized token

 show expected token

9.2.7.2.4 Layouts

If TextTransformer changes between editing and debug mode, the layout is changed automatically too. If you don't change the according options, the default layouts are used respectively. On the *Layouts* page of the options dialog for the user interface you can choose other layouts.



All layouts, which are stored in the *Settings* folder of the DATA FOLDER, are listed in the selection boxes on the right. You can choose one of them as well for the edit mode as another for the debug mode.

If the box *Auto-save layouts* is checked, a changed layout will be saved automatically if the mode of the program is changed or if it is closed.

9.2.7.3 Environment options

On the first register page of the environment options a tree is presented. The nodes of the tree show the sections of the Ini-file *tetra.ini*, which exists in the same directory as the TextTransformer program: *tetra.exe*.

CONFIG
EXTENSIONS
FRAMES
PATH

On the second register page you can change the set of filters for the files, which can be opened by the TextTransformer.

9.2.7.3.1 CONFIG

Here you can choose the language of the of the user interface, by setting

English for English (default)
German for German

9.2.7.3.2 EXTENSIONS

Here you can choose the extension for the files, which are created from the TextTransformer

Key	Meaning	Default value
Cpp_Header_Extension	extension of the created header files	h
Cpp_Source_Extension	extension of the created source files	cpp
ComponentSupport_Extension	extension of the file, created for the component support	pas

9.2.7.3.3 FRAMES

Here you can choose the names of the default frame files. These files are used for the generation of code or for the component support.

Key	Meaning	Default value
Cpp_ParserHeader	Frame for Header-files	ttparser_h.frm
Cpp_ParserSource	Frame for sourcecode-files	ttparser_c.frm
ComponentSupport	Frame for the component support file	enums_pas.frm

Note: For each project, you also can create individual frame files, which are stored in the according project directories.

9.2.7.3.4 PATH

The TextTransformer program is installed into the PROGRAM FOLDER. When you start the program for the first time you are asked, to select a DATA FOLDER for the user data.

Here you can choose different paths:

Frames default "DATA FOLDER\Frames"

Directory for the default frames for code generation

Projects default "DATA FOLDER\Projects"

The project directory is the root directory of all your different TextTransformer projects.

Backup, default: "DATA FOLDER\Backup"

Into the backup directory the group of files is copied, which shall be saved before they are transformed. This directory can be modified temporarily.

Target, default "DATA FOLDER\Target"

The target directory is set at first as the target for the transformation of groups of files. This directory can be modified temporarily.

9.2.7.3.5 File filter

On the second register page of the environment options and in the dialog for the transformation of groups of files you can change the set of filters for the files, which can be opened by the TextTransformer.

In the box all types of files are listed, which you can choose at the moment, either to open a file into the input window or to filter a group of files for transformation. Each item can be deleted or changed.

To add a new filter, you first have to click on the *New* button. Now you can write the description and the filter itself into the according fields. It is possible, to list several masks into a single filter. To do this, you have to separate the single masks by semicolons. For example:

Description: Pascal files
Filter: *.PAS;*.DPK;*.DPR

If a mask was added, deleted or changed and you finish the dialog by *Ok*, the whole list is saved, so that you can use it in the next session with the TextTransformers again.

9.2.7.4 Project options

The project options are valid for the actual project.
They are stored together with the project automatically. But you also can save and load them individually as ASCII file. For this the dialog of the project options has its own menu. Project option files have the extension: **tto**.

The options are place on several register pages:

Names and Directories
Parser/Scanner
Inclusions
Encoding
Warnings/Errors
Code generation

9.2.7.4.1 Names and Directories

On the first page of the project options names and directories can be selected. Directories are calculated relatively to the project.

Start rule
Test file
Frame path

9.2.7.4.1.1 Start rule

A start rule can be selected from the list of all productions, which is set, when you open the project.

If no start rule is selcted here, the production with the same name as the project will be set as start rule. If there is no such production, no start rule will be set.

9.2.7.4.1.2 Test file

It is possible to select a file which is loaded into the source text window when opening a project. This is primarily desirable as long as the project is under the development.

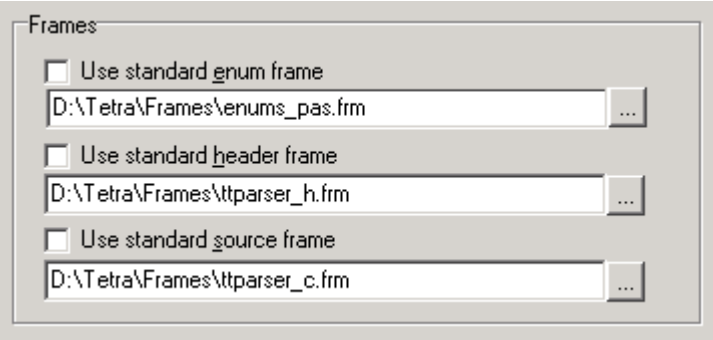
9.2.7.4.1.3 Preprocessor

Source files can be preprocessed while they are loaded. The preprocessor is another instance of the TETRA interpreter, which executes a TETRA project. This project can be set here.

Another preprocessor project also can be set in the preprocessor project. So a whole batch of pre-processing can be executed.

9.2.7.4.1.4 Frame path

Here you can set the paths, where the frame files are stored, which are used for the creation of code.



Default frames are in the FRAMES directory.

enums_pas.frm is the default frame for the component support
ttparser_h.frm is the default frame for the c++ header
ttparser_c.frm is the default frame for the c++ code

These standard frames can be modified individually for each project. It is recommended to save the modified frames together with the project into a common directory.

9.2.7.4.2 Parser/Scanner

The following options can be set as project options for parsers and scanner.

Ignorable characters
Case sensitivitiy
Word bounds
Parameter and {{..}}
Global scanner
No failure alternative to SKIP
No failure alternative to ANY

9.2.7.4.2.1 Ignorable characters

Usually spaces within the source text of a program are irrelevant, and when, the TextTransformer looks for the start of a token, it will simply ignore them. Other separators like tabs, line ends, and form feeds may also be declared irrelevant.

Example:

A Production for the sum of two terms can be easy written as:

Sum = Term "+" Term

This rule not only shall recognize a text like:

"23+4"

but also

"23 + 4" and " 23 + 4" etc.

The spaces between the numbers and the plus operator are irrelevant and should be skipped. If the space were not set as irrelevant, additional token for the gap between the terms and the operator had to be defined. For example:

Space = "[\n\r\t]*" (linefeeds line breaks, tabs and spaces).

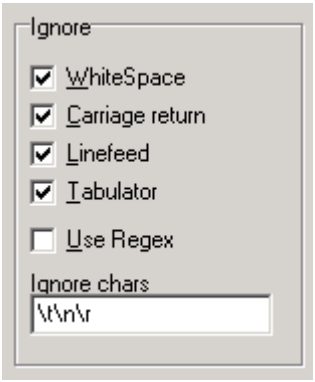
The production above had to be reformulated:

Sum = Term Space "+" Space Term

Depending on the activation of the check box *Regex*, the ignorable characters will be defined as a list of characters or as a regular expression.

Definition of the ignorable characters as a list of characters

Spaces, linefeeds, line breaks and tabs are set as ignorable characters per default. They can be removed or added to the list of ignorable characters simply by clicking the accordant check box.



Manually other character can be added too.

Definition of the ignorable characters as a regular Expression

In spite of a list, you can define the ignorable characters also by means of a regular expression. To do so, the **box Regex** must be activated. The text of the edit field now will be interpreted as a regular expression.
For example the expression "\s*" could be set. Then all characters of the character set \s would be skipped. That's about the same as a character list, where all check boxes are activated. An example

that makes more sense is:

(\s|//[^\r\n]*)*

By means of this expression not only the spaces will be skipped, but also line comments.

You also can set the **name of an already defined token** into the edit field. Now this token defines the ignorable characters.
If the check box regex is activated and the text in the edit field only consists of literal characters, the text will be interpreted as the name of a token.

Remark: A regular expression, that defines the ignorable characters, will automatically be included into parenthesis and preceded by the anchor "\\A", to assert, that the skipped section always will begins at the actual text position.

Remark: When you use a list of ignorable characters, it is possible to access the skipped characters, which follow on a SKIP node, by xState.str(-1). If you use a regular expression, this is not possible.

9.2.7.4.2.2 Case sensitive

If the option *CaseSensitive* is activated upper case letters are distinct from lower case letters.

9.2.7.4.2.3 Word bounds

This option only applies to the recognition of literal tokens, but not on regular expressions.
If the word bounds option is activated, a token will be recognized only, if it begins or ends with a word bound. A word bound mostly is the transition of an alphanumerical character or the underscore and a character which doesn't belong to this class \w. Exactly a word bound is defined by three cases:

The character adjacent to the token is not member of \w
The exterior character of the token doesn't belong to \w
The token is situated at the begin or the end of the input

Example:

In the text:

"sindbad the seaman",

following expressions have two word bounds: "sindbad", "the" and "seaman".
Only one word bound is in: "bad", sea" and "man".

At this example you can see, that it is possible to analyze the internal structure of single literal words, if you deactivate the word bounds.

Normally it is recommended to activate word bounds, because otherwise there is a great danger of

wrong recognitions. For example: if word bounds are deactivated and the token "end" is defined, the beginning of the name of a variable in the following line is wrong recognized:

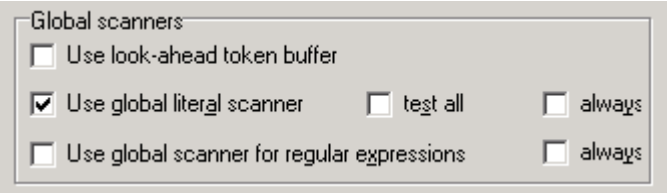
```
  endVar := 10;
end
```

9.2.7.4.2.4 Parameter and {{...}}

Here you can determine, if not bracketed text of the parameter field or the field for actions, which accompany token, is interpretable or exportable or both, that means, if it will be executed internally or copied to the generated code. The same option determines the treatment of actions, which are included into double braces.

9.2.7.4.2.5 Global scanner

On the page *Parser/Scanner* of the project options three check boxes exists, by which you can make some fine-tuning of the scanning process.



It is recommended, to leave the default settings, with activated global scanner for literals. If you don't have special reasons to change the scanning process, you can leave the default settings and you don't have to read the following explanations.

The buffering of the look-ahead tokens can accelerate the execution of a project if makes heavy use of look-ahead's. This isn't the case mostly.

The other options in this box control of the sets of token which are tested respectively in the current position of the grammar. The speed of the parser and the error probability are influenced by these sets. Larger token sets slow down the parser and increase the probability that a token is found that doesn't match to the grammar. The latter may be wanted in certain cases, e.g. inside of a look-ahead.
If **no global scanners** are used, then only the tokens are looked for, which just can follow in accordance with the grammar. If however global scanners are used, then **all literals** can be tested **always** or they are tetsted only,if a at least one literal can follow according to the grammar. Similar for the **regular expression**: either all are testet **always** or all are tested, if at least one can follow.

Note: Internally there is a third Scanner, which can be global or local. It's the **scanner for ignorable characters**. Whether this scanner is local or global is determined by the setting of ignorable characters in the local options.

This once again with some more details:

There are three steps of evaluating the next token corresponding to the three kinds of scanners. Beginning at the actual position in the input has to be evaluated

1. whether ignorable characters are following, and then
2. whether a literal token is following or,
3. whether a token defined by a regular expression is following

These three tests can either be done by a single global scanner, or by local scanners. The use of one global scanner is the traditional method, applied by all parser generators hitherto. The use of local scanners is based on the idea, to test only those candidates for the next token, which are part of the actual alternatives. For example: to test the following structure it is necessary to decide, whether an a or a b token exists at the actual position

(a | b) c d

A local scanner will test exactly this. A traditional global scanner in contrast will test all token of a grammar, that means at least a, b, c and d token. The result will be the same for both types of scanner. The difference lies in the speed and the expense. When using local scanners the speed will be higher, but a bigger amount of storage is needed.

The result of the text analysis also can be influenced by the choice of global or local scanners. By use of a global scanner the probability of conflicts between different token is greater than by use of local scanners, where only a little set of tokens compete with each other. This is the reason why there is the additional possibility to **limit testing on the currently expected tokens** even if a global scanner for literals is used.

Example:

Text:	"int int"
Produktion:	"int" ID
Token ID:	\w+

If all literal tokens are always tested, the second occurrence of "int" isn't recognized as an ID. The literal token "int" is rather recognized once more. So the text cannot be parsed. This is desired if the text e.g. is C++ code. A variable may not have the name of a variable type.

Text:	"Sir Sir"
Produktion:	"Sir" NAME
Token NAME:	\w+

Look into the phone book and you will find the name "Sir". So the salutation "Sir Sir" is definitely correct. It only is recognized if you don't test on all literal tokens.

Rule of thumb:
All literals should be tested for formalized languages with defined key words at significant positions. Only the expected literals should be tested otherwise The local options also can be adapted respectively, if necessary.

Conflicts, which can result from the use of a global scanner, can be the cause of error messages

like

Matching but not accepted token: ...

At the description of this message an example is presented.

A further example of the effects of the scanner options is given at the explanations for the look-ahead production.

By buffering of the look-ahead tokens parsing can be accelerated if look-ahead productions are used. If at first a look-ahead is tested in a production, then the tokens found are put on a stack and read from there again if the initial production is further executed.

Example:

```
Prod1  ::= IF( Prod2() ) "a" "b" ...
Prod2  ::= "a" "b"
```

If *Prod2* has been tested, the tokens "a" and "b" are already anylsed from the text.

This method can work only, if some restrictions of the otherwise existing possibilities are accepted. E.g. the following then doesn't work any more:

```
Prod1  ::= IF( Prod2() ) ID+ ...
Prod2  ::= "a" "b"
```

It can be recommended often, to use ANY instead of SKIP. A sequence of tokens recognized with *ANY+* in a look-ahead is reproduced correctly as a sequence of the specific tokens which were subsumed under ANY when reading the stack.

In the semantic actions sub-expressions cannot be accessed from the tokens fetched from the stack. Inclusions cannot be used in combination with the look-ahead buffer.

9.2.7.4.3 Start parameters

The parameters which are provided to the IDE for the functions *ConfigParam* and *ExtraParam* can be entered here in two fields. When the project is executed via the transformation manager, in the command line tool or as generated code the parameters set here are ignored and the parameters of the appropriate surroundings are used instead.

9.2.7.4.4 Inclusions (comments)

Here you can select a production from a box, which shall be used for parsing inclusions - mostly comments. If this production is put in the global project options, then it is valid for all productions of the project, i.e. also for the chosen production itself; so e.g. nested comments are parsed correctly.

Example:

```
CppComment ::= "/*" ( NUMBER | ID | "." | "-" )* "*/"
```

If *CppComment* is set in the project options for parsing inclusions, the following comment is also parsed:

```
/* 1. level /* 2. level */ 1. level */
```

Unfortunately, the following production doesn't work if comments are nested:

```
CppComment ::= "/*" ( SKIP | STRING )* "*/"
// ! this definition isn't appropriate for nested comments
```

The Text, recognized by SKIP were:

```
1. level /* 2. level
```

So the beginning of the inner comment would be skipped and the end of the inner comment then interpreted as the end of the complete inclusion.

If the nesting isn't wished, then in the local options of *CppComment* the empty field can be set for the inclusions. As well for any other production the inclusions can be adjusted individually.

9.2.7.4.5 Encoding

For the settings of this register page some restrictions have to be taken into account. **The settings cannot be reproduced completely in the debug mode** and the functions to write UTF-8 encoded output, only are implemented in the transformation manager of the TextTransformer and in command line tool and are not part of the code accompanying the Professional Version of the TextTransformer.

For the source text and the target text you can adjust the encoding independently:

ANSI-Text
UTF-8

You cannot use RedirectOutput, if UTF-8 options is set. By this option RedirectOutput is performed already.

Further you can adjust for the input file and the destination file, in which mode they are opened:

Text mode
Binary mode

If files are read and written in the **binary mode**, then there is no difference of their data in the working memory and on the hard disk. In the **text mode**,however, conversions are made under Windows for the treatment of the line breaks. So, e.g. a single linefeed characters '\n' is combined with a carriage return character '\r' to"\r\n" when the file is written..

Example:

After execution of the statement:

out << '\n'; or out << endl;

"\r\n" is written in the output text.

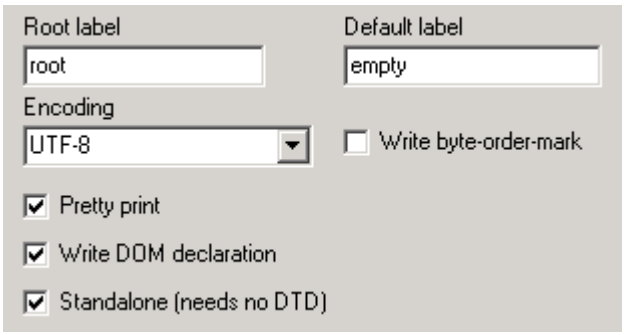
In the **binary mode** such a transformation isn't carried out. A Windows line break then must be written explicitly by:

out << "\r\n " or out << '\r' << endl;

The editor - in contrast to the viewer - is not able to represent line breaks with simple linefeed characters.(However, both types of line breaks can be recognized by the EOL token.)

9.2.7.4.6 xerces DOM

The options which have to be set for a standalone XML document are in the upper half of the dialog page. Such a document doesn't depend on a DTD.



The shown options produce a document, this looks as follows:

```
<?xml  version="1.0"  encoding="UTF-8"  standalone="yes"  ?>
<root>
...
</root>
```

Root label

Here the root tag is defined. In the example it is called "root".

Default label

Each tag of an XML document must have a name. This name is the label of a *dnode*. To keep the analogy to the construction possibilities of node's, *dnode*'s can be used without explicitly defined labels. Such *dnode* then gets internally the default label assigned automatically.

```
dnode dn(". "text");
```

then appears with the default label above as:

```
<empty>text</empty>
```

Before a parser is called in the generated c++ code, the default label has to be set. *CTT_DomNode* has a static method for this purpose.

```
dnode::SetDefaultLabel(L"default_label");
```

Encoding:

Xerces supports many encodings for the output of the XML documents, which are discussed below. However, only the ANSI character set (Windows 1252) is represented correctly in the working surface of the TextTransformer. ANSI or a different 8 bit encoding should therefore be used in the developmental stage of a project. Otherwise the text appears in the output window:

Encoding cannot be written into the output window of the IDE

If a project is executed by the transformation manager, the command line tool or the Delphi components, there isn't any restriction for the encoding.

Here some remarks copied from

http://xerces.apache.org/xerces-c/faq-parse.html

concerning the different supported encodings

ASCII

ISO-8859-1 (aka Latin1)

For UNIX systems in countries speaking Western European languages, the encoding will usually be iso-8859-1

Windows-1252

The default character set on Windows systems is windows-1252 (ANSI), not iso-8859-1. While Xerces-C++ does recognize this Windows encoding, it is a poor choice for portable XML data because it is not widely recognized by other XML processing tools.

UTF-8

UTF-8 - like UTF-16 - covers the full Unicode character set, which includes all of the characters from all major national, international and industry character sets. This encoding - like UTF-16 - is more widely supported by XML processors than any others
Efficient. utf-8 has the smaller storage requirements for documents that are primarily composed of characters from the Latin alphabet.

UTF-16 (Big/Small Endian)

UTF-16 - like UTF-8 - covers the full Unicode character set, which includes all of the characters from all major national, international and industry character sets. This encoding - like UTF-8 - is more widely supported by XML processors than any others

UCS4 (Big/Small Endian)

EBCDIC code pages IBM037, IBM1047 and IBM1140 encodings
(Extended Binary Coded Decimals Interchange Code)

IBM1140
IBM037
IBM1047

When creating EBCDIC encoded XML data, the preferred encoding is IBM1140. The IBM037 encoding, and its alternate name, ebcdic-cp-us, is almost the same as IBM1140, but it lacks the Euro symbol.

Write byte-order-mark (BOM)

At some encodings a byte order mark (BOM) can be set at the beginning of a file.the mark tells in which order the bytes must be evaluated, if single characters consist in several bytes, like in UTF-16.

The BOM is written at the beginning of the resultant XML stream, if the output encoding is one of the following:

- UTF-16
- UTF-16LE
- UTF-16BE
- UCS-4
- UCS-4LE
- UCS-4BE

Such a mark also can optionally be used in UTF-8 to indicate the file as UTF-8 encoded. However, it isn't possible to set a BOM for UTF-8 per option automatically, since xerces doesn't support this. However, the BOM can be written by the following code in the program:

```
out << char(0xEF) << char(0xBB) << char(0xBF);
WriteDocument();
```

If a UTF-8 encoded file is read as an ANSI file, this mark appears as: ï»¿.

Encoding	Bytes
UTF-8	EF BB BF
UTF-16 Big Endian	FE FF
UTF-16 Little Endian	FF FE

Pretty-print

This formats the output by adding a newline carriage return and indented whitespace to produce a pretty-printed, human-readable form.

If this option is set and the document shall be written with a *WriteDocument* without parameters, it is required for some encodings - e.g. for UTF-16 - to set the option for a binary output too.

Write DOM-declaration

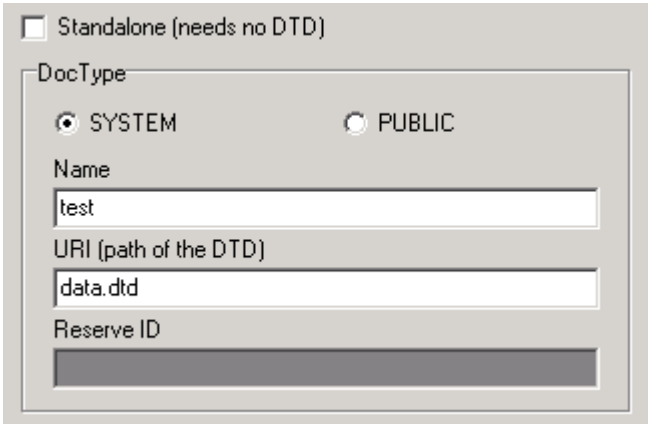
The line in the example above::

```
<?xml  version="1.0"  encoding="UTF-8"  standalone="yes"  ?>
```

is the declaration of the document. If it shall not be written or be formulated explicitly in the program - for UTF-8 only -, then the checkbox can be deactivated.

9.2.7.4.6.1 DTD

The options which have to be set for an XML document, which depends on a DTD (document type definition) are in the lower half of the dialog page.



The shown options produce a document, this looks as follows:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE test SYSTEM "data.dtd">
<root>
...
</root>
```

With the DTD "data.dtd" is checked, whether the document is valid. However, this examination will be done when the document is reloaded in a validating XML processor and is not carried out in the TextTransformer.

If a public DTD shall be used, the corresponding radio button has to be selected and the lower field for the reserve ID is activated. A reserve ID is required.

9.2.7.4.7 Warnings/Errors

The creation of warnings can be suppressed or activated.
This concerns the warnings:

node is nullable
node is start and successor of nullable structures
xState parameter for member functions
System overlap

Stack maximum

9.2.7.4.7.1 Stack maximum

The **first value** refers to the stack of the productions of the main parser.
By this value the internal stack will be limited. This limitation is a protection against infinite loops, which can appear in case of left recursions. The default value 100 normally suffices.

Remark: the internal stack is greater than the shown stack, because the internal stack contains in addition branches to sub-rules.

The **second value** refers to the stack of the productions of a look-ahead. The limitation of this stack is important to avoid a crash of the system at possible circular calls.

9.2.7.4.8 Code generation

These options only are interesting for user of the professional version.

const

Wide-Character-Regex
Only copy all code
Characters and increment of indentation
Plugin type
Template parameter for plugin character type

9.2.7.4.8.1 const

const

The activation of this option only makes sense, if you want to produce c++-code with the professional version of the TextTransformer. In the interpreted applications of the standard version the option implies unnecessary restrictions of your programming possibilities.

If the const-option is activated, all functions (productions, token actions and interpreter functions) of the created parser become const functions.

Example:

```
void CCalcParser::Expression(double xd) const;
```

this assures, that these functions will not change the data of the parser class. This is important e.g., if you are writing a multi threaded application.

If the const-option is activated, all operations are forbidden, which will change the state of the parser class. This includes operations, which will change the position of the cursor of an *mstrstr* class variable (findKey, gotoNext etc.). But it is possible, to use mstrstr-references as class variables instead. Also the source and target directories and the indentation stack only can be used as according references. The frame for the class must be adapted accordingly.

9.2.7.4.8.2 Use wide characters

If the wide-character option is activated, parser classes are created, which can process texts, which are using character sets with more than the 256 ASCII characters (Unicode). A single character then is represented by two bytes in the memory instead of only one. Instead of *char*, *std::string* and *boost::regex* the according data types: *wchar_t*, *std::wstring* and *boost::wregex* are used.

When this option is used, the parser class in the generated code is derived from:

```
CTT_Parser<wchar_t>
```

All texts and string functions of the parser systems then are based on the *wchar_t* type.

However, the "diagnostic" system isn't changed by the *wchar_t* template parameter : Error texts still are strings from *chars* and Meta functions like *ProductionName* still return std::string. If the diagnostic system and the parsing system are mixed, it can happen that the generated C++ code doesn't compile because of incompatible string operations.

9.2.7.4.8.3 Only copy all code

If this option is set, the parts of the code in the semantic actions, which are written into a generated c++ parser, simply are copied.
The option normally isn't set. The interpretable parts then are reconstructed from the parsed form and there will be a new formatting and a transformation of some constructs. In this case the exportable parts also are changed.
Such transformations are necessary for

- member function calls, which need an additional *xState* parameter
- Abbreviated notations, which are only valid in the interpreter, are replaced: out -> xState.out(); indent -> xState.indent(), format -> boost::format.
- insertions (by *add*) of member functions into function tables. Here the second string parameter - the name of the function - is translated to a pointer of the member function.
- If a wide character parser is generated, some additional replacements are done: "Hi" -> L"Hi"; format -> boost::wformat;

9.2.7.4.8.4 Characters and increment of indentation

Indentations in the generated parser code can be alternatively made by blanks **ws** or by tabulators **tab**. The number of these characters, which are inserted at the respective positions, determines degree of the the indentation.

9.2.7.4.8.5 Operating system

For the Windows operating system the generated code files can be written with "\r\n" line breaks and for Unix systems with "\n" line breaks.

Note: this option has no influence on the behavior of the parser.

9.2.7.4.8.6 Plugin type

You can define a plugin for a parser. The plugin can be initialized outside of the parser and accessed as well inside the parser as outside after the parsing has finished. For example you can store the results of the parsing inside of the plugin.
By the plugin dynamic data can be used even in a const parser for multithreading applications.
The pointer to the plugin type is set into the class for the parse state. Per default this is

CTT_ParseStatePlugin

If dnodes are used in the project, you have to chose

CTT_ParseStateDomPlugin

In this case the Xerces library has to be linked to the produced code.

If this type needs a template parameter for the character type, this can be set by the next option.

If no such pointer is passed to an interface-method, a local instance of the plugin will be created automatically. CTT_ParseStatePlugin contains all data, which are necessary for the plugin-methods.

User defined plugins have to be derived from CTT_ParseStatePlugin or from CTT_ParseStatePluginAbs.

The pointer to the plugin-type is a template parameter for the parse state class and is also written as a typedef into the parser class. So the complete type of the plugin in known inside of the parser and data and functions of the plugin can be accessed without a typecast.

9.2.7.4.8.7 Template parameter for plugin character type

If this option is set, the template parameter "< char_type >" will be added to the plugin type in the generated code. If for example the default plugin type is set:

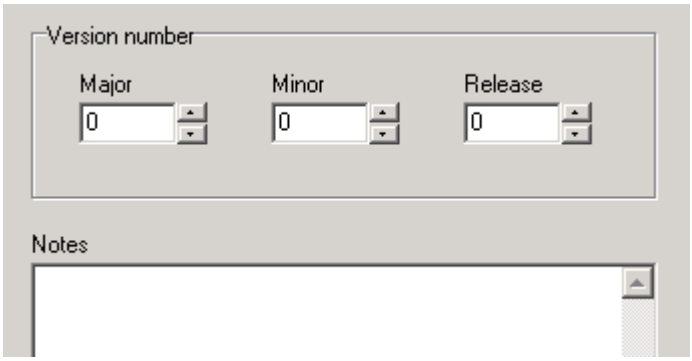
CTT_ParseStatePlugin

this will be completed in the generated code to:

CTT_ParseStatePlugin< char_type >

9.2.7.4.9 Version information

On the page with the title *Version* you can save information to the version of your project.



9.2.7.5 Local options

The project options, which are valid for all productions at first, can be overwritten for single productions. A menu item *Local options* exists in the main menu *Project*, which is shown only, if you are on the production page.

9.2.7.5.1 Local options

The project options, which at first are valid for all productions, can be overwritten for single productions. A menu item *Local options* exists in the main menu *Project*, which is shown only, if you are on the production page. The local settings of a production don't have an effect on the operation of the productions called in it.

Before you can set the different items, the local options must be activated on the first page of the local options dialog. If local options are activated, which overwrite a project option, this will be show in the syntax tree after compiling by a red hook:

Remark: There also exist local options, which don't overwrite project options.

In the local options the option for the use of the global scanner is moderated

Global scanner for Literals, if possible
Global scanner for Regex, if possible

Activation of the global scanner will have an effect only, if the CaseSensitive option in the project and the local options are the same. Local deactivation of the global scanner is possible every time.

If options for the scanner are changed locally, then this change has an effect as soon as within the production a new token will be looked up. This happens as soon as a token is accepted in the production or, if the last token of a production called in the production is accepted. In the last case the successor of the called production is flooked up in the calling production.

Example:

```
Prod1 ::= Prod2 "c"
Prod2 ::= "a" "b"
```

If in *Prod1* the blank is locally set to be ignored, but not in *Prod2* and if in both the word separation is deactivated, the first row of texts is parsed correctly and the second isn't parsed. Whether the third text is parsed depends on the previous production.

1: "ab c", "abc" abc ", ab c "
2: "a bc"
3. " abc"

Remark: The **ignored characters** are accumulated by all tests. If no successor of a token is found in the actual rule, the successor of the rule itself will be looked up. But the look up will start after the characters already ignored.

Besides the items described at the project options, here exists one more option:

Create interface

This option can be set also directly from the tool bar of the production page

If this option is activated, a special scanner will be created, which can make a test of all those tokens, which are contained in the first set of the rule.
If there are several compiled productions, for which interfaces are created, you can interactively change between them while transforming one text. After such a change, the interface scanner tests, whether the new start rule matches the actual text.

Remark:
If the productions are compiled by Parse all scripts, this has the effect as if there were set the interface option for every production.

Normally a parser is called by the interface to its start rule. But it is possible also to call other productions directly, if the according interface method is created.
For example the TextTransformer itself parses the parameter of a production by a sub rule of the production parser.

9.2.8 Menu: Windows

A lot of information is represented in in the TextTransformer, distributed over a variety of docking windows. Depending on the actual working step and the kind of the project a certain part of it is needed. The optimal order of the windows depends on the technical equipment (e.g. the screen resolution) and the individual preferences of the user. Therefore there is the possibility to make and store special window layouts, to have them at the disposal again, when required.

For the basic situations of editing and debugging projects there are prefabricated default layouts

EditDefault.ds
DebugDefault.ds

If TextTransformer changes between editing and debug mode, the layout is changed automatically too. On the Layouts page of the options dialog for the user interface you can choose other layouts for the two modes of the program.

The menu items in detail are:

Window list
Customize layout
Save layout
Restore default edit layout
Restore default debug layout
Windows

A video, that demonstrates th adaption of a layout, is at

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

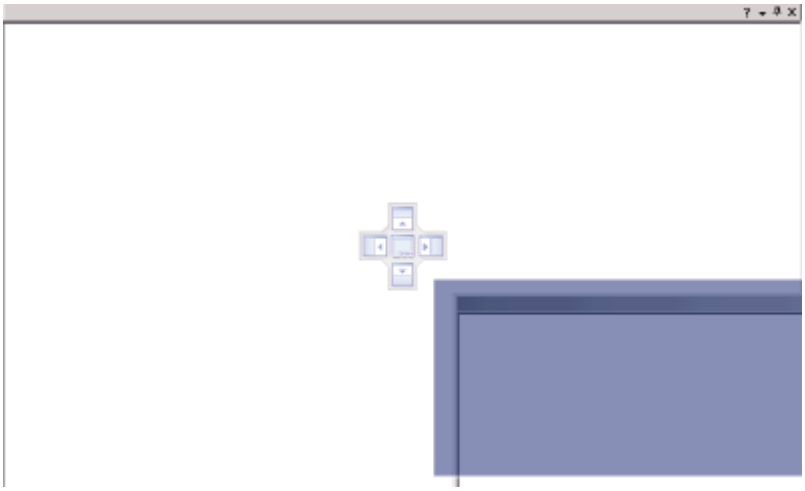
9.2.8.1 Docking Windows

You can customize panel layouts in TextTransformer and create custom docking schemes. This feature is especially convenient for smaller display resolutions, since it allows you to display several panels simultaneously.

The relative size of the windows mostly can be changed with the mouse. If the mouse is positioned over the borderline of the two blocks the mouse pointer changes its form to

While pressing the left mouse button you can drag the line to the new position, where it remains, if you let off the button.

To undock a panel, simply double-click or drag the panel's caption. This will turn the panel into a normal, floating window. You can quickly dock a floating panel back to its previous location by double-clicking the panel caption. If you want to change the panel location, drag it to the desired place. When you are dragging the undocked panel over another panel, TextTransformer will show the docking zone selector, which lets you specify where the panel will be docked if you release it:

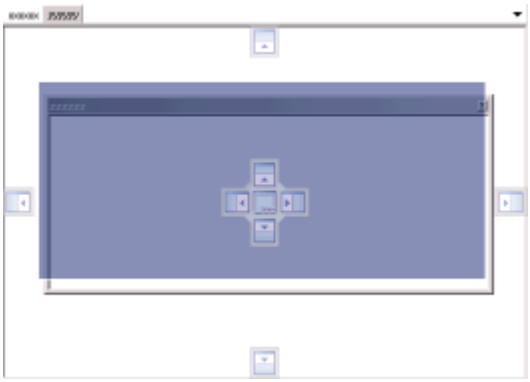


To dock a panel to the left, top, right or bottom edge of the the lower panel, move the mouse cursor to the  ,  ,  or  icon within the Selector and then release the mouse button.

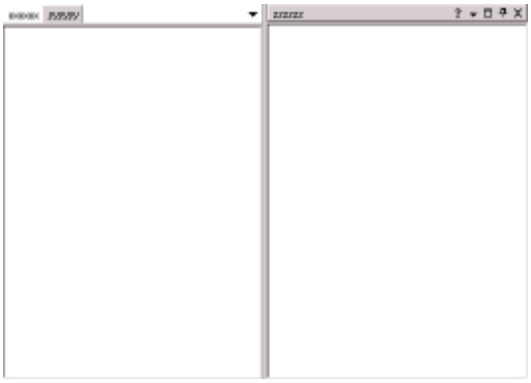
Once a floating panel has been docked onto another panel, it still has a caption bar, with a name at the left and buttons at the right, and the previous contents likewise have a caption bar on top of their section of the page. Any section can again be turned into a floating window by dragging the caption.

If you select the  icon in the docking zone selector, the floating panel and the lower panel will be docked at the same spot. In this case, the panels will be organized as tabbed pages (tabbed panels are simply panels docked not inside one another, but at the same spot). You can dock more than two panels to the same spot. To undock a tabbed panel, just drag or double-click its caption.

Note that when you are dragging a float panel over a tabbed one, besides the docking zone selector in the center of the tabbed panel Automated Build Studio will also show docking zone selectors along the edges of the tabbed page.



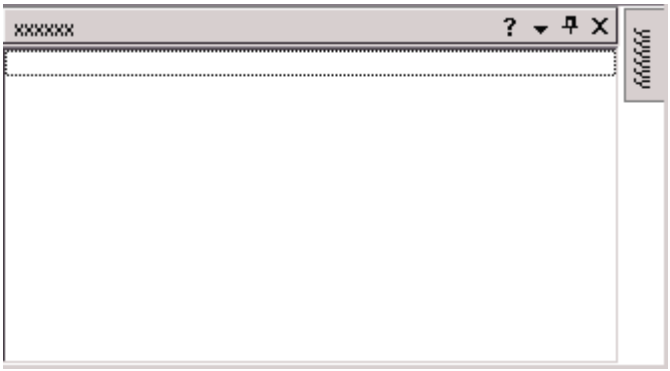
These selectors let you dock the floating panel relative to the whole tabbed spot. Using the central selector you will dock the floating panel to the the tabbed panel that is currently in front of the other tabbed pages.



If you select this variant, then the tabbed page will hold two panels.

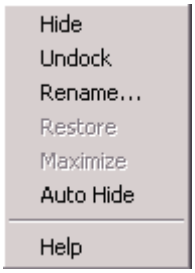
A docked panel can be auto-hidden. Auto-hide means a panel can be minimized along one of

the sides of the docking site, so only the panel's caption remains visible. For instance, on the following figure the "yyyyyy" panel is auto-hidden:



To hide a panel, click the button in the panel's caption.
To view an auto-hidden panel, move the mouse cursor to the panel caption or click the caption and the panel will pop up.
To show a hidden panel, simply click the button in the panel caption.

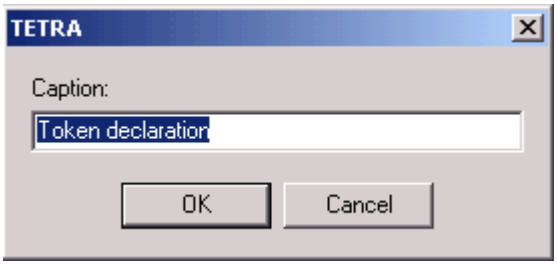
Right-clicking a tab shows the context menu with the following items:



Rename Calls the Panel Caption dialog where you can rename the tab.
Help The Help item is displayed if the tabbed page contains one panel only. It calls the help topic with the panel description. If the tabbed page holds several panels, the Help item is hidden.

9.2.8.1.1 Caption Dialog

The **Panel Caption** dialog is used to change the caption of a panel or a tabbed page (if several panels are docked to the same spot, they are displayed as tabbed pages). To call this dialog, right-click the caption of the desired docked panel or tabbed page and select **Rename** from the context menu. To call the Panel Caption dialog for an undocked panel, right-click its caption and select **Docking | Rename** from the resulting context menu.

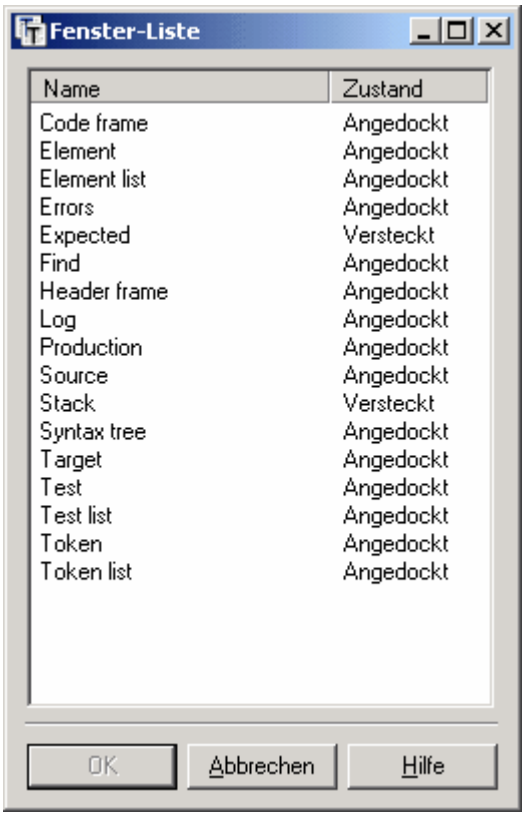


See also
Docking Windows

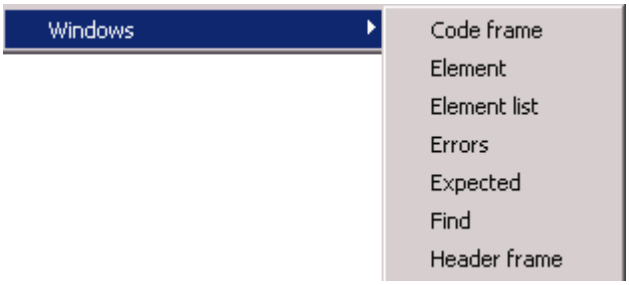
9.2.8.2 Window list

You can navigate by the window list fast to windows which are hidden or closed currently. The window list exists in two kinds:

1. as a dialog which can be called also with the button  in the tool bar.



2. as an item of the Windows menu, which can be accessed very quickly.



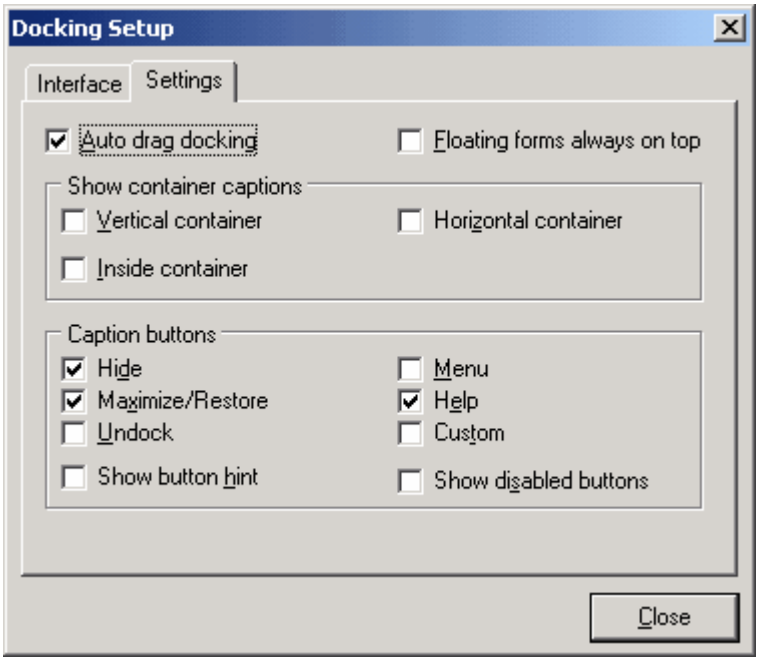
If you click on an item in the list, the according window will be shown.

The windows are:

Code frame
Element
Element list
Errors
Expected
Find
Header frame
Log
Production
Source
Stack
Syntax tree (= production list)
Target
Test
Test list
Token
Token list

9.2.8.3 Customize layout

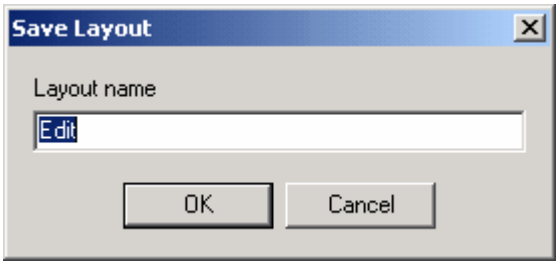
The dialog for customizing the layout is available only in the professional version and only with English labels. With the dialog e.g. moving groups of panels is made easier as additional common captions can be displayed . The additional elements only apply to the current session and aren't stored.



9.2.8.4 Save Layout

The current layout can be saved by the item *Save Layout* in the Windows menu or by the button  in the toolbar..

At first a dialog appears, where you can enter a name for the layout. The name of the current layout is set as default name.



If you confirm, the layout will be saved into the Folder DATA FOLDER\Settings with the extension ".ds" appended to the name.

All layouts in DATA FOLDER\Settings are shown in the selection box in the toolbar on the left side of the save button_

If you select one of the items in this box, the according layout is loaded and the windows are arranged accordingly.

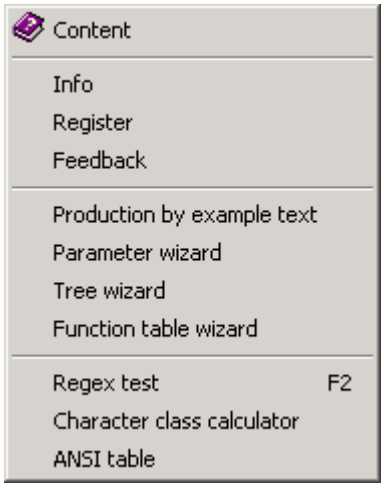
9.2.8.5 Restore default layout

At the installation of TextTransformer the layouts EditDefault.ds and DebugDefault.ds are written into DATA FOLDER\Settings. These layouts are made for a screen resolution of 800 x 600 pixels and should be adapted to the technical conditions and to the needs of the user. To be able to experiment with these layouts safely there is a safety copy in programPROGRAM FOLDER\bin. By the two items of the main menu Windows :

Restoring default editing layout
Restoring default Debug layout

the safety copies can be copied back automatically into the *Settings* folder.

9.2.9 Menu: Help



By the menu *Help* you can call this help and the Registration dialog. In addition two tools and some wizards can be accessed:

Regex Test
Character class calculator
ASCII table

The help for single instructions can be obtained directly by selecting the according word and pressing F1.

```
{{
str s;
vstr vInherited;
}}
```

A help to error messages can be obtained directly by selecting the according line in the error list and pressing F1.

9.2.9.1 Feedback

If you have problems, comments or ideas, your feedback is welcome. Please don't hesitate to send a mail to

dme@texttransformer.com

It is recommended to use this frame for your mail, because it contains important information about the your version of the TextTransformer and the system you are using. Please write your remarks into the following lines and then copy the whole text into your e-mail program.

Your feedback only will be used for an improvement of the TextTransformer. Your information will not be passed on a third party and you will not receive unwanted advertising mails. This feedback will be answered adequately.

9.2.9.2 Wizards

Some wizards make the work with TextTransformer simpler. Refraining from the wizard for new projects, they can be called by the Help menu.

New project wizard
Creating a production from an example text
Parameter-Wizard
Tree-Wizard
Function-Table-Wizard

The wizards offer a number of options on every of their pages, which are explained on the upper edge of the page. Depending on the selection made, further pages can become obtainable, to which can be gone with the ***Next*** button.

The operations of wizards cannot be undone! Please save your project before using the wizard and reload the project, if an error occurred.

The use of the Tree wizard and of the function table wizard is described in detail for the Java parser

9.2.9.2.1 New project wizard

When creating a new project a wizard appears, which helps to make some basic settings, depending on planned project type. This wizard even can create complete projects for small tasks, which can occur in your daily work.

The following project types can be chosen:

Multiple replacement
  Multiple replacements of words
  Multiple replacements of characters

Rewriting lines
  CSV-wizard
  Creating a line parser from an example text

Header/Chapters/Footer

New project from scratch

The example texts and projects used on the following pages can be found in the directory:

..\TextTransformer\Examples\Wizards

9.2.9.2.1.1 Multiple replacements of words

By the TextTransformer projects can be made which work similarly as the search-and-replace function in word processing programs. But with the TextTransformer a lot of files can be processed at once and many substitutions can be carried out at the same time in each of these files. The wizard described here supports the construction of such projects.

To this you simply have to input the list of searching and replace expressions into a table.
The input of the values into the table works just like at the tables of other wizards.

If you have entered all word pairs and have operated the confirming button by which the editing mode is exited, you get the text of a production displayed on the next page of the wizard. This production suffices for the whole word substitution project.

```
(
    "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:]]+
```

If you now finish the wizard, the project is ready for application.

The substitution list can be stored and afterwards reloaded. Frequently, it will be easier to carry out modifications and complements directly in the created project, however.

In the directory:

..\TextTransformer\Examples\Wizards

there are two lists of titles of films in English and in German, which you can use for experiments

Film_en_ge.txt und Film_ge_en.txt

9.2.9.2.1.2 Multiple replacement of strings

The multiple replacement of strings is quite similar to the multiple replacement of words. The difference is, that parts of words or other text sections also can be replaced in a project for multiple replacement of strings. But the number of possible substitutions is limited here. (A number below hundred should not make any problems.)

Example of a created production:

```
(
    "Ha"     {{ out << xState.str(-1) << "He";}}
  | "elt"    {{ out << xState.str(-1) << "orld";}}
  | SKIP     {{ out << xState.copy(); }}
)*
```

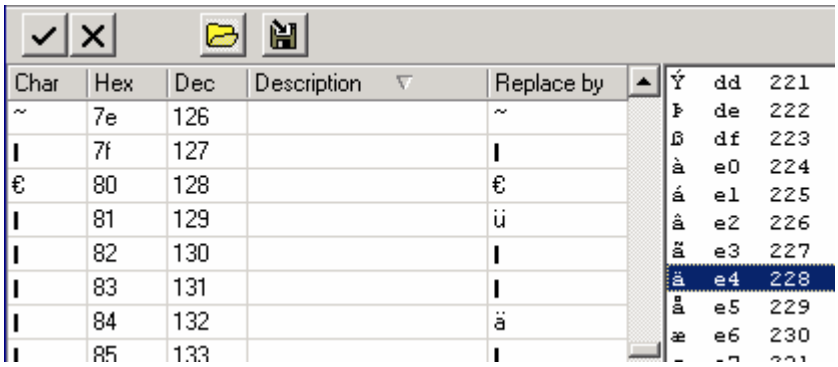
The word bound option is dactivated here.

9.2.9.2.1.3 Multiple replacements of characters

By the character substitution wizard you can create projects, which replace a number of letters by other characters or literals. E.g. this can be necessary for the conversion of text files, which were written on another operating system.

A character substitution project is very similar to a word replacement project. However, you don't have to write the substitution list, since there is only a restricted set of characters. You simply can select for a character in the table another one from a list. Characters, for which no replacement is chosen, will be copied unchanged into the target text. Instead of selecting a replacement character in the right box, it is possible too, to write a literal replacement expression directly into the according field of the table. By this, the table is set into the edit mode.

Select a character in the table **first**. As soon, as you select a character from the right box, the text in the actual row of the table will be overwritten.



The content of the column: "Replace by ...", can be saved and reloaded later.

If in all rows of the table the desired assignements of characters are set and you have operated the confirming button by which the editing mood is exited, you get the text of a production displayed on the next page of the wizard. This production suffices for the whole character substitution project. For example the production for the conversion of an Atari text looks like:

```
(
    "□"     {{ 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(); }}
)*
```

If you now finish the wizard, the project is ready for application.

In the directory:

..\TextTransformer\Examples\Wizards

there are two lists of characters:

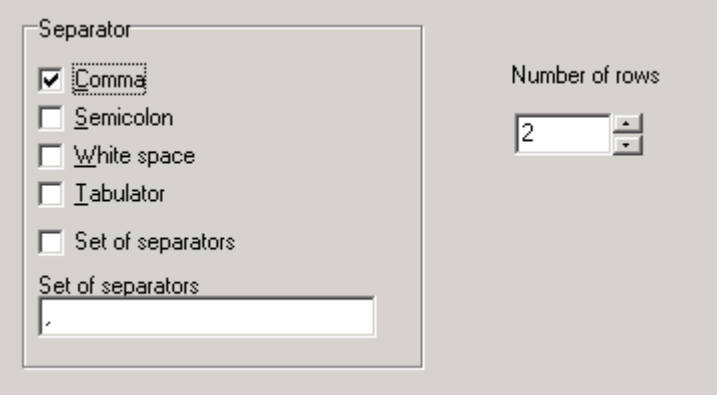
ANSI2DOS.txt und DOS2ANSI.txt

by which you can create projects for the conversion between the ANSI character set and the DOS character set.

9.2.9.2.1.4 CSV-wizard

With the abbreviation *CSV* (Character Separated Values) files are named whose lines consist of data, which are separated by commas or other separators from each other. Many database applications can read and write such files.

The wizard described here, allows extracting the individual data. You then can change them or arrange them differently. It is assumed that the separator doesn't occur within the data. If this shouldn't be the case at your CSV file, then you can use the wizard, which generates a line parser from an example text.



It's possible not only to define a comma as a line separator, but any arbitrary other character too. It is also possibly to define a set of separators, but two fields always will be separated by only one of them. If more than a character separates the columns, you can define the separator also as a literal expression.

After the number of columns was set, you can go to the next page of the wizard, where you can choose the kind of actions you want to be generated. On the next page then you can see the text of a production, which is generated from the settings.

For writing a simple comma separated text of two columns in string variables the production is:

```
{{
str sCol1, sCol2;
}}

(
SKIP     {{sCol1 = xState.str(); }}
","
SKIP     {{sCol2 = xState.str(); }}
EOL
)

{{
// out << Here you can output the columns in the desired form.
}}
```

E.g. the comment in the second to the last line could be replaced now by:

```
out << sCol2 << "," << sCol1 << endl;
```

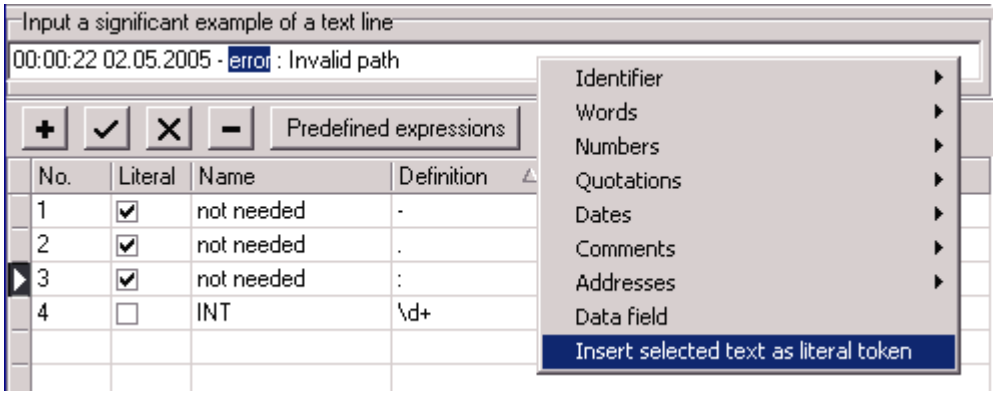
An output text would be created, in which the columns of the source text would be exchanged.

A video, which demonstrates this wizard, is at:

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

9.2.9.2.1.5 Creating a line parser from an example text

If a file consists of lines which all have the same structure - e.g. a log file -, you can create a complete parser for this file with the wizard described here.



At first you have to copy a typical line into the edit field above the table. Then you have to enter the definitions of the tokens into the table, by which the line shall be analyzed. This input is very simple for literal expressions. There is the menu item in the pop-up menu, which appears after clicking with the right mouse button: Insert the select text as a literal token. With this function a text that was selected with the mouse in the edit field can be inserted directly as a token into the table. Otherwise the input of the values into the table works just like at the tables of other wizards.

If you choose the direct output for the actions on the next page of the wizard, you will get the following **Chapter** production, after you have finished the wizard:

```
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
```

This production consists of the tokens of the table in the order as they are retrieved from the

example text. Text parts that aren't recognized by the tokens of the table will be recognized with the SKIP token.

9.2.9.2.1.6 Header/Chapters/Footer

A frequent general structure of texts is the subdivision in a header, some chapters and footer. For example in a book, the contents would be the header and the index would be the footer. The wizard described here generates a frame for such a structure. For each: the header, the chapters and the footer, a production will be created, which are called from the start rule.
At first only the **SKIP** symbol is used in the **header** and the **footer** production. The exact structure of these parts must be written by hand.

It is possible for the **chapter** production to define some tokens succeeding one another in the wizard already.

The input of the values into the table works just like at the tables of other wizards.

If the token by which the chapter production starts is specified obviously so, that it cannot occur in the header part, then executable parsers can already be created with the frame created by the wizard.

Example

To process the keywords in a HTML file, you can "travel" across significant expressions in the file up to search position.

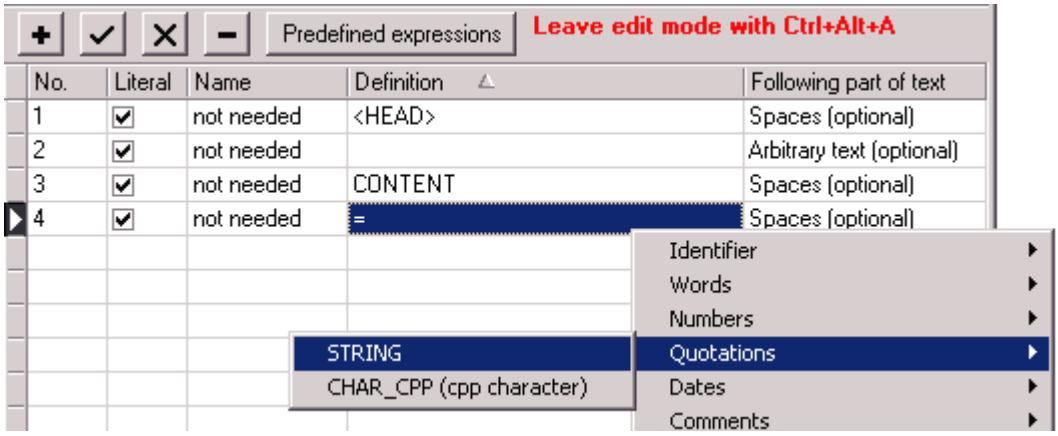
The shortened beginning of the file:

C:\TextTransformer\Examples\Assistenten\textkonverter.html

looks like:

```
<!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>
...
```

If a sequence of tokens is defined as presented in the picture:



the text will be skipped by the **Header** production to the expression "<Head>", then the expression "\"keywords\"" is searched, then "CONTENT" and "=" and finally the searched string will be found. The **Footer** production will skip the rest of the text.

If you choose the direct output for the actions on the next page of the wizard, you will get the following **Chapter** production, after you have finished the wizard:

```
"<HEAD>"     {{out << xState.copy(); }}
( SKIP     {{out << xState.copy(); }} )?
"CONTENT"     {{out << xState.copy(); }}
"="     {{out << xState.copy(); }}
STRING     {{out << xState.copy(); }}
```

Here you easily can change the action, which is combined with the *STRING* token according to your plans.

9.2.9.2.1.7 Actions

For the parsers created by the wizard, actions can be generated automatically, which consist in copying the recognized text mostly. These actions then can be modified with little effort, so that the output text gets the desired form.

Direct output

For every token an action is created, which writes the recognized text section together with the ignored characters directly in the output. E.g.:

```
Token1 {{ out << xState.copy();}}
Token2 {{ out << xState.copy();}}
```

The direct output is the most efficient way to process the text and therefore should be chosen if possible. The order of the text sections stays with it unchanged, though.

Writing into string variables

For every token an action is created, which writes the recognized text section together with the

ignored characters into string variables. E.g.:

```
Token1 {{ s1 = xState.copy();}}
Token2 {{ s2 = xState.copy();}}
```

The recognized text sections are duplicated here, and then can be written in an arbitrary order, however. E.g.:

```
out << s2 << s1;
```

Creating a parse tree

For every token an action is created, which writes the recognized text section together with the ignored characters into node variables. E.g.:

```
Token1 {{ nRule.add("Token1", xState.copy());}}
Token2 {{ nRule.add("Token2", xState.copy());}}
```

A parse tree allows versatile and multiple further processing of its nodes. It, however, isn't trivial, to write correct routines for this processing.

Creating a DOM

With this option a parse-tree is produced like above, but from dnode's instead of from node's. A XML document finally is written in the output.

No actions

Is then advisable to write the code for the actions by hand if only a small portion of the recognized text shall be put out or if the above methods shall be combined with each other.

9.2.9.2.2 Creating a production from an example text

By this wizard you can create a linear production from an example text. The production will consist of a simple sequence of tokens, which arises from the order in which the tokens are found in the text.

This wizard is very similar to the wizard, which creates a line parser from an example text.

There are two differences:

1. The example text must not be a line. If you select a part of the source text in the input window before opening the wizard, this part will be taken as example automatically.

2. Tokens, already defined in the actual project, will be inserted into the table of the tokens used to analyze the example text. You can delete the tokens from the table, which shall not be tested.

9.2.9.2.3 Parameter-Wizard

The parameter wizard simplifies the creation of a uniform parameter or a uniform variable declaration for several scripts.

Short explanations of the offered options are presented on the individual pages of the wizard. Therefore here only briefly is outlined, what can be achieved with the wizard as a result for all productions and tokens at the example of a single production. The production:

```
A ( ) ::= _a ( A | B )
```

The production can be equipped with a parameter *xParam* of the type *type* and a local declaration of such a parameter variable.

```
A ( type& xParam ) ::=

{{
type  Param;
}}
_a[Param]
(
     A[Param]
   | B[Param]
)
```

If the option for the creation of declarations isn't set, you get:

```
A ( type& xParam ) ::=

_a[xParam]
(
     A[xParam]
   | B[xParam]
)
```

If e.g. xParam is of the type *str*, these scaffoldings then can be completed easily so that the reference variable xParam contains the desired target text after passing through the complete parser. For the named literal _a the addition could look like::

```
_a( str& xParam ) ::=

{{
   xParam += xState.copy();
}}
```

If for all tokens corresponding actions were defined, then xParam would include a copy of the source text after processing the parser.

The automatic generation of the code - in the example "[xParam]" - to pass the parameter to the called productions and tokens is only possible if the option: "for all productions and tokens" is chosen.

The tree wizard works quite similarly as the parameter wizard for the special type node.

9.2.9.2.4 Tree-Wizard

The tree wizard simplifies the creation of tree nodes for several scripts. In contrast to the parameter wizard, the tree wizard can create code too, which inserts the node parameters into a whole tree. For complete projects this wizard also can insert actions for the creation of nodes for the pure literal tokens and the wizard can pass the nodes as calling parameters to the called productions and tokens.
All this is useful for the generation of parse trees and their evaluation by function tables. For the latter there is an extra wizard. In the help for the function table wizard there are some additional explanations of the background for the work with parse trees.

If you choose the option, to create nodes inside of productions, for the example of a single production:

```
A ( ) ::= (_a | "b" ) ( A | B )
```

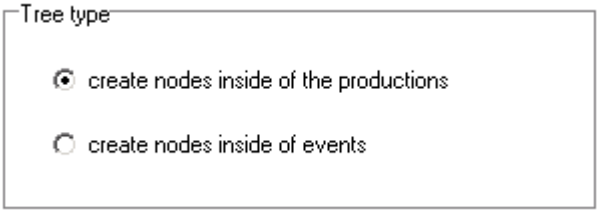
the result, that is produced, if the option "for all productions and tokens" is chosen, looks like:

```
A ( node& xn ) ::=

{{
node  n("A");
xn.addChildLast(n);
}}
(
    _a[n]
   | "b"   {{n.add( "LITERAL", State.str() );}}
)
(
     A[n]
   | B[n]
)
```

The automatic generation of the code - in the example "[n]" - to pass the node to the called productions and tokens is only possible if the option: "for all productions and tokens"or "complete" is chosen.

9.2.9.2.4.1 Tree type

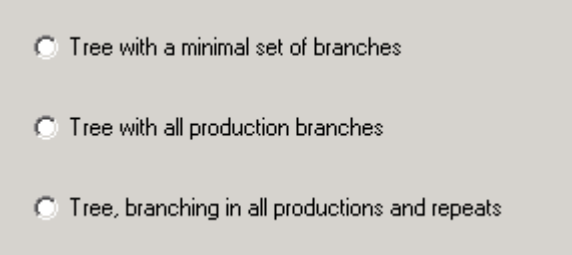
If you call the tree wizard in the HELP menu, then a choice appears for the manner, how the tree shall be created.



The operation of this wizard is explained in detail in the Java example.It is the advantage of this option that by changes of the inserted code, the tree creation can be controlled easily. So e.g. specifically certain elements can be ignored.

It is the advantage of the tree creation by the events however that the tree generation is independently carried out from the code of the productions.

There are three kinds of trees which can be generated:



On the one hand, they are different by the number of elements represented in them and on the other hand by the time of the creation. The tree with a minimal number of branches is produced from a help container after the last production called from the start production is left. The other trees are produced during the the parsing process and can already be used then.

There is a video, which demonstrates this kind of tree:

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

9.2.9.2.5 Function-Table-Wizard

The function table wizard exists in a small version and in an extended version. The small **Quick-Wizard** for the extension of an existing function table with single functions appears, if you click with the right mouse button on the name of a function table in the list of the class elements. The **extended wizard**, which is described below, can be invoked either from the quick wizard or as an item of the help menu.

By means of the function table wizard new tables can be created and existing tables can be extended, whereby it is possible to create whole groups of correspondences of labels and function names at once. In cooperation with the tree wizard it is even possible to insert code for the generation of a complete parse tree for a complete project in all productions automatically. This possibility should be executed only once for a project since a repeat leads to name conflicts.

The names of the created functions and the name of the function table are derived from a base-name:

If the beginning "m_ft" of the name for the function table is changed afterwards. the wizard wiil not work correctly with this table any more.

Background:

One of the paradigms for transformation programs to create in a first step a parse tree which then can be used in various ways for the generation of output. Tree nodes in the parse tree represent productions and tokens.
The function table wizard is supporting a recommendable scheme to design such trees in the TextTransformer. In accordance with this scheme the names of the respective productions or tokens are used as labels of the nodes.
Special tables can be created in the TextTransformer: the function tables, which assign functions to the labels of the nodes. This relation is thus indirectly at the same time a relation of functions to the productions or tokens. Each of these functions serves for the processing of the accompanying node and with that for the processing of the production or the token.

Different tokens or productions frequently are treated in the same way. E.g. many of the texts recognized by tokens have to be output again unchanged. Different tokens or productions can therefore be processed with the same function. If a special token requires, however, a special treatment, then a special function is written for its node.

Example:

Name of the script	**Label of the node**	**Function**
normal1	normal1	Handle_Default
normal2	normal2	Handle_Default
special1a	special1a	Handle_special1
special1b	special1b	Handle_special1
special2	special2	Handle_special2

The inititialization of the function table *m_ft* looks like:

```
m_ft.add("", "Handle_Default");
m_ft.add("special1a", "Handle_special1");
m_ft.add("special1b", "Handle_special1");
m_ft.add("special2", "Handle_special2");
```

The corresponding nodes must be created and added to the complete parse tree in the scripts. A reference on the last node of the parse tree is submitted to the script as a parameter:

```
Parameter:
        node& xNode
Text:
        node n("normal1");
        xNode.addChildLast(n);
```

or

```
Parameter:
        node& xNode
Text:
        node n("special2");
```

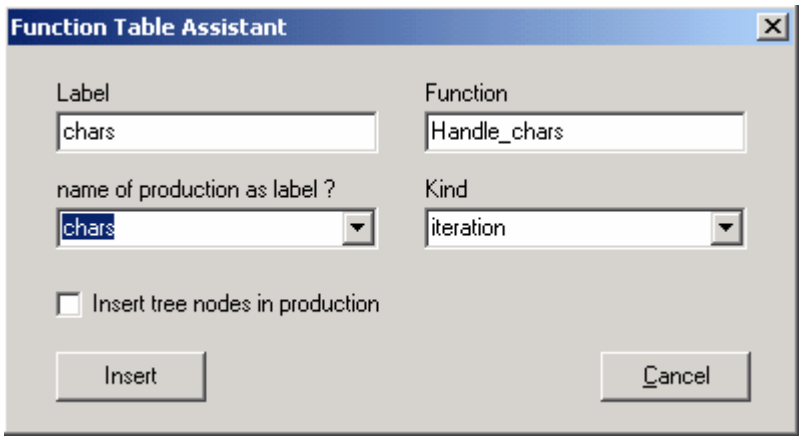
```
xNode.addChildLast(n);
```

Accordingly for the other scripts.
When a script is called, an additional parameter has to be passed now:

```
normal1[n]
special2[n]
```

9.2.9.2.5.1 Quick wizard for function tables

The function table wizard makes the extension of a function table with new functions easier. The wizard appears if you click with the right mouse button on the name of a function table in the list of the class elements.



Write the label of the node type and the name of the function, which shall handle this type, in the upper two fields. Then, by the ***Insert*** button an additional entry for the function table is created:

```
m_ftExpr.add("chars", "Handle_chars");
```

and at the same time a new function is created, which has the same parameters as defined for the function table and a return type, which corresponds to the type of the function table.
In the combo box ***Kind***, you first can chose a frame type for the new function.
If e.g. the actual function table is of the type *str_mstrfun* with a node parameter, with the ***iterator*** frame the following function will be inserted on the element page:

Name: Handle_chars
Type: 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;
}}
```

For the *Kind* ***value*** the following text is created:

```
{{
 return xnNode.value();
}}
```

For the *Kind* ***empty*** the brackets are created only.
The function frame then can be modified by hand.

A simple method to construct a solid linkage of:

1. a function table
2. an according function
3. the nodes, which represent the production (the text recognized by the production)

is, to name the label by the name of the production. That's why the wizard contains a list of the **names of the productions as choice for the label**.
If the check box ***Insert tree nodes in production*** is activated, in front of the parameters of the production is put an additional node parameter and to the beginning of the text the following code is written:

```
node n("chars");
xNode.addChildLast(n);
```

For the calls of the production the additional parameter must be inserted by hand.
If in the calling production the same code was written by the wizard, the additional parameter mostly is: n:

9.2.9.2.6 Input tables

There are tables in several wizards for the input of expressions. The operation of these tables is always the same.

There are four buttons in the tool bar:

 Insert a new row

 Accept the value of the edited field

 Cancel changes in the edited field.

 Remove row

Edit-mode

After a new row is inserted into the table, the wizard is put into an editing mode, in which some keyboard inputs have changed function. So the work is simpler in the table. The table is in the edit mode now and you can write directly into the fields. The input can be accepted by the accept-button or by the enter key. By the arrow keys you can navigate in the table then and with **Strg+Enter** you can append a new row.

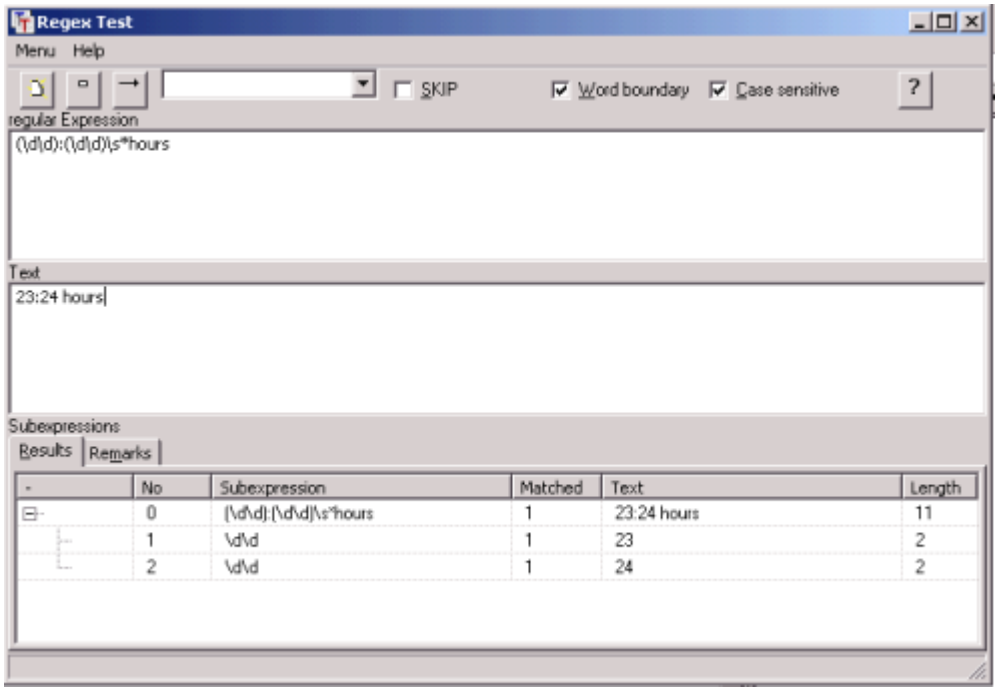
The edit mode is left by the *accept*, *cancel* or *delete* button or by pressing the *Esc* key **once**. If you would press Esc a second time, the wizard would be closed. Attention: the Enter key also get back it's old meaning, when the edit mode is left: it starts the function of the current control element.

Literal

There is a column in some tables with the heading "literal". If the check box is activated in this column, then the token doesn't need a name and the definition text is taken in literal meaning. That means, the characters which have a Meta-meaning in regular expressions, don't have these here: every letter means itself. A backslash doesn't have to be put in front of the quotation mark and the backslash here as in the case of the definition of literal tokens directly within a production either.

9.2.9.3 Regex test

An item of the *Help* menu is *Regex Test*. Here you can call a dialog box for testing single regular expressions.



The dialog consists in a menu, a tool bar and three sub-windows.
The items of the menu can be executed by means of the buttons in the tool bar too.

Clear fields

The contents of the three windows are cleared.

Compile

The regular expression in the topmost window will be compiled. Depending on the correctness of the syntax the success will be confirmed or an error will be shown.

Execute

The regular expression in the topmost window will be compiled and applied to the text in the middle window. If the expression is syntactical correct, it will be listed together with its sub-expressions in the undermost window. In the right of each sub-expression the sections of text is presented, which is matched by the sub-expression. In the example shown, the first sub-expression matches the hours and the second the minutes of the time.

List box

In the list box of the tool bar all tokens of the actual project are listed. If you choose one of these tokens, its definition will be shown in the sub most window of the dialog box.

Options

SKIP

The option SKIP is usually deactivated. Then a match in the text only is found, if it matches at the start of text. (When parsing a text the next token shall match at the current text position normally.) If the option SKIP is activated, then in the complete text is searched for to the next position where the regular expression matches. **If the text beginning matches, this is judged to be a fault**.

Word boundary

The word boundary option here has the same effect for literals as the word boundary option in the project settings. The evaluation is done as in the case of the parsers produced by TETRA, by a special ternary tree. **For regular expressions this option doesn't have any consequences here** , since the expression is taken unchanged from the topmost edit field The SKIP expressions of a production are created however, while compiling the project and literal sub-expressions then are inserted together with word boundaries expressed by the anchors "\<" and "\>".

Case sensitive

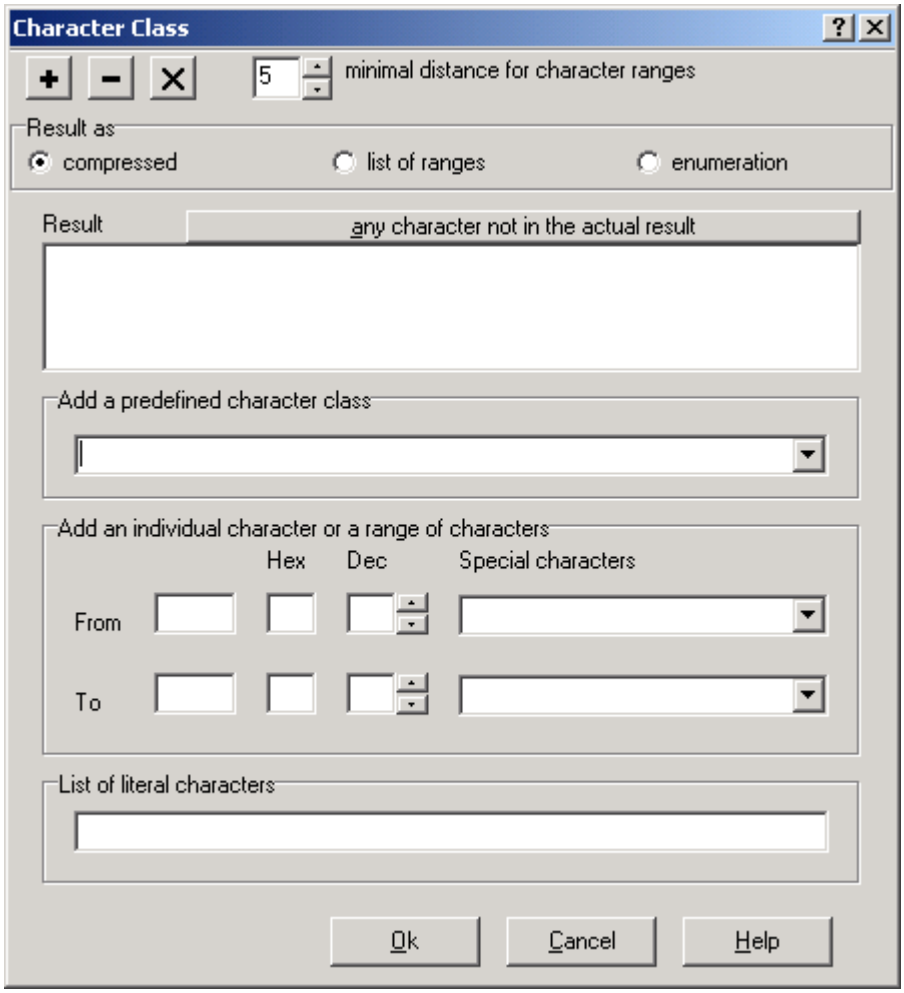
The case sensitive option here has the same effect as the case sensitive option in the project settings.

Remark:

The test of the regular expression works in the same manner as a scanner of a TextTransformer project. If the expression in the up most window is a literal, this will be mentioned in the result window. The expression then will not be evaluated as regular expression, but as a literal. While "" and '\"' are indifferent for a regular expression, this is not the case for the test of literals.

9.2.9.4 Character class calculator

The character class calculator is made to construct character classes step by step, by adding or subtracting characters from the actually resulting class.



The characters, which shall be added or removed, can be selected from a list of predefined character classes or they can be put in as a range or a list of characters or an individual character.

The same rules as for the characters of a string apply to the literal character list. E.g. line breaks '\n' also can be inserted. The backslash character must therefore also be doubled: \\, to insert it into a list.

The button marked with '+' in the tool bar of the character class calculator

is used to add characters to the resulting class. The characters to be added must have been selected before by means of the dialog elements below.

The button marked with '-' in the tool bar of the character class calculator

is used to remove characters from the resulting class The characters to be removed must have been selected before by means of the dialog elements below. If characters are selected which don't exist

in the result class, then this doesn't have any consequence.

The button marked with 'x'

deletes the previous selection.

The **result** can be represented **as**

compressed

Characters are summarized into character classes provided that the set is complete. The remaining characters are represented as ranges or single characters.

list of ranges

Characters succeeding one another in the ANSI table are summarized to ranges. E.g. the range of 1-6. results from 1,2,3,4,5,6. The remaining characters are listed one by one.
Whether the characters are put into a range expression or not also depends on the following setting

minimal distance for character ranges

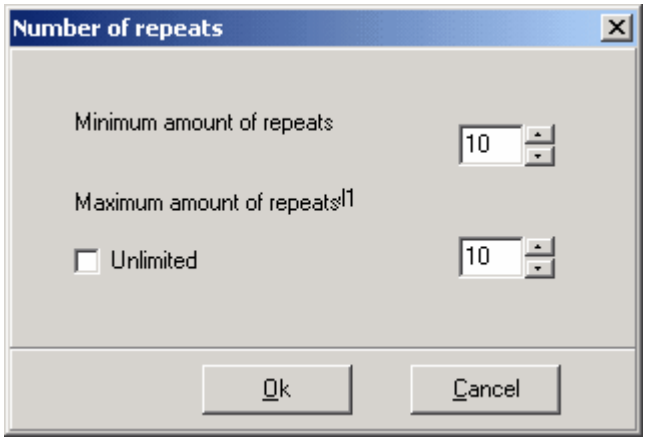
Characters are put into a range expression only when in accordance with the ANSI list at least so many other characters are situated between the smallest and the greatest character as the minimal distance is demanding.

enumeration

All characters are listed one by one..

With the longish button above the result list: ***all characters, which are not in the actual result***, you can invert the previous result. So all characters are listed, which were not selected before.

If the dialog is exited with Ok, a second dialog appears, where you can enter the number of characters, which shall follow each other.



After pressing *Ok* a second time, the resulting expression will be copied into the clipboard.

Example:

The Wikipedia dokumentation :

http://en.wikipedia.org/wiki/Wikipedia:How_to_edit_a_page

defines:

In the URL, all symbols must be among:
A-Z a-z 0-9 . _ \ / ~ % - + & # ? ! = ( ) @ \x80-\xFF

To construct this character set, you have to:

1. add the character set: "any word character - all alphanumeric characters plus the underscore"

2. add the character range from Hex 80 to Hex ff

3. add the character list: ".\\/~%+-&#?!=()@"

Caution: the list may not contain white spaces and the backslash has to be doubled.

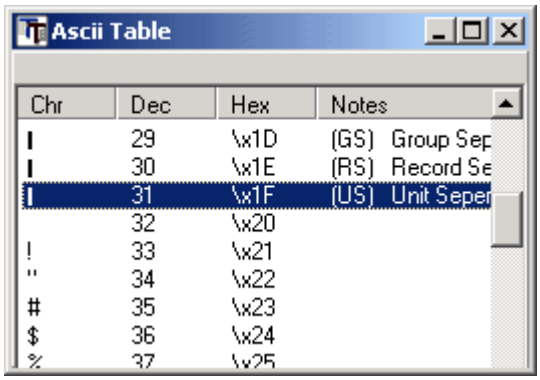
If you are pressing Ok now and let the unlimited repeat, the result is:

```
[-[:word:]€-ÿ!#%&()+./=?@\\~]+
```

The hyphen appears in front of the expression, as it shall not characterize a range.

9.2.9.5 ANSI table

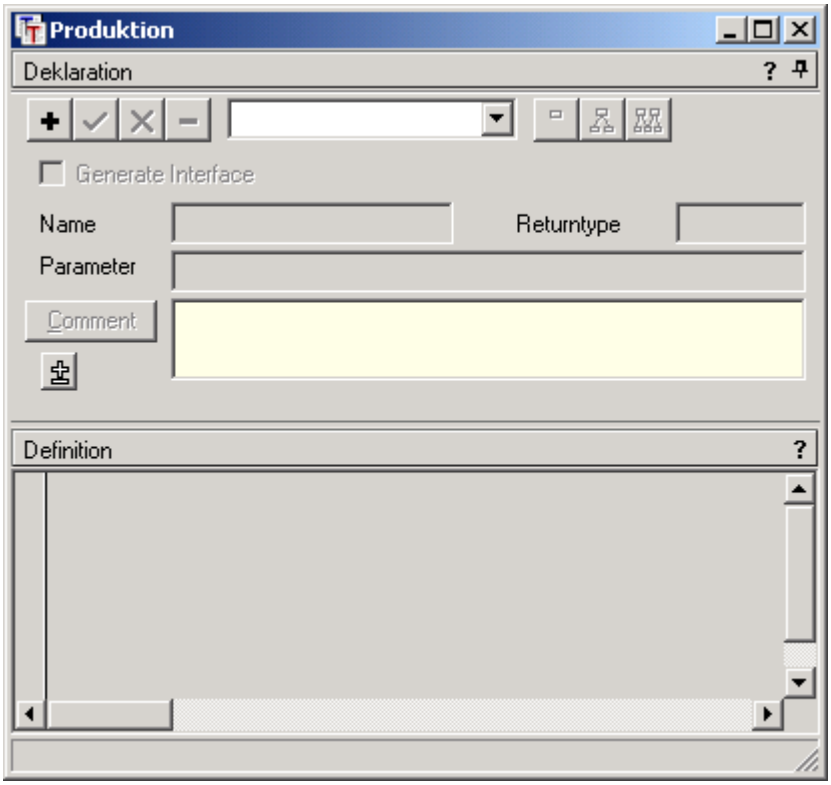
At the item *ASCII-Table* of the *Help* menu a list of all ASCII characters is shown.



In the first column the character itself is shown, as far as it is depictable. The second column contains the according code in decimal presentation and in the third column in hexadecimal notation. In the last column there are additional remarks for some characters.

9.3 Script management and parsing

There are four similar windows for the four types of scripts. In each window one script is shown and can be edited. The illustration shows the production window as an example:

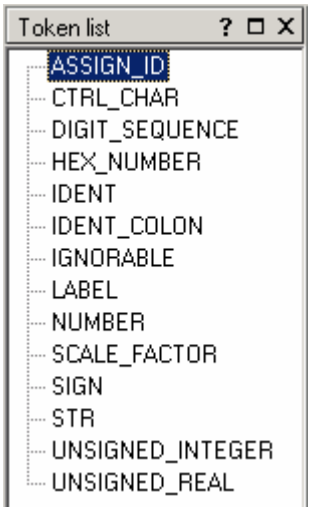


There is one such window each for:

1. Token
2. Productions
3. Class elements (variables and functions)
4. Tests

These four pages have a uniform toolbar and a corresponding menu, by which tokens, productions, functions, variables or tests can be changed, renamed, deleted or added.

There is another window each for the lists of all scripts of a type available in a project. If one of the names is selected in the list, then the corresponding script is shown. The illustration shows the token list as an example:



A special case is the **list of the productions**. It is shown on the syntax tree window. For compiled productions not only the names of the production but their complete syntactic structure is represented here.

In a new project without any rules nearly all buttons are menu items are disabled.
A first rule can be inserted by clicking on the insert button or by means of the menu item: Project | New.

9.3.1 Tool bar and menu

All script pages have a same looking tool bar.

 New script

 Accept changes

 Cancel changes

 Delete script

 Parse/Test single script

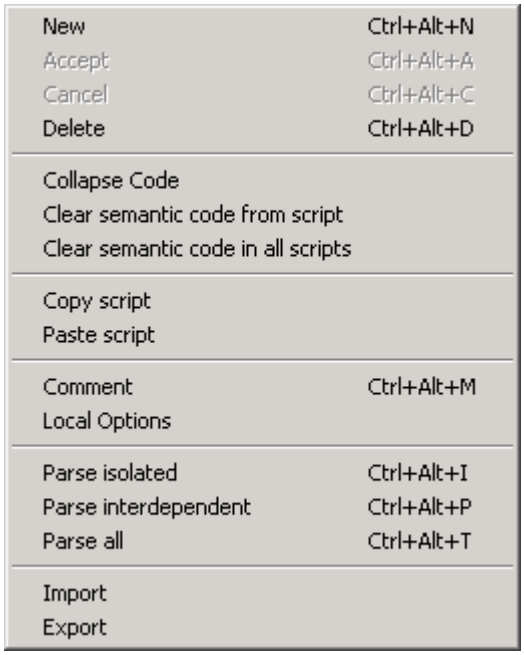
 Parse/Test all connected scripts (with a common start rule or group)

 Parse/Test all scripts

 to the previous script

 again to script

The same actions are accessible as items of the project menu. This menu is displayed only inside of the main menu, if a tab window for a script management is visible.



In the project menu additional items for local options, for import and export of scripts and to remove the semantic code in scripts are shown. On the element page and on the test page instead of the last menu item, there is the possibility to erase all scripts of the corresponding page.

9.3.2 Insert

 New script

To create a new script you can use the +-button. The gray edit fields will turn to white and text can be written into them. At least a name and a text must be written; otherwise the script will not be accepted. A name may contain the alphanumerical character and the underscore, in which the underscore may not be the first character.

Examples.: Identifier, Const_declaration, UB40

Each new name must differ from all other names by at least one character.

9.3.3 Delete

 Delete script

To delete a script, it first must be chosen from the list of all scripts. Now you can click the button [-] or the according item in the menu.

You can't undo the delete function. After you have saved the project all deleted scripts are lost.
As long as the project wasn't saved, you can reload the old project to restore the deleted scripts, but then you will loose all other changes. In the case of emergency you can open a second instance of the TextTransformer and copy the script from there.

9.3.4 Edit

An existing script can be edited immediately. Simply write the new text into the according field. By this the TextTransformer automatically changes into the edit mode.

If a script is in edit mode is shown at the background color of the edit fields.

Background: white	=	edit mode
gray	=	not in edit mode

As soon as one of the fields of a script was changed, also the state of the tool bar buttons will change: the insert and the delete button are disabled, while the accept button will be enabled.

9.3.5 Cancel

 Cancel changes

By this button or the corresponding menu item you can reset a changed script to its previous state, as long as the changes were not confirmed.

9.3.6 Accept

 Accept changes

A new script or a changed one will be taken over into the repository, by confirming the changes with the accept button or the corresponding item in the menu.

The background color of a script editor is white if it is in the editing mode. As soon as the changes are accepted, the background color changes to gray.

9.3.7 Rename

A script can be renamed. It first must be chosen from the list of all scripts. Now the name can be changed in the according edit field. After confirming with

 Accept changes

the script will be taken over into the repository, if the new name doesn't collide with an existing name, i.e. if it is different from all other names. Otherwise there will be a warning.

9.3.8 Navigation

To choose a script, click on its name in the list of all scripts.

A double click inside the text of a production on the (bold printed) name of a different production will display it.

By means of the buttons in the tool bar

 to the previous script

 again to the script

You can navigate to the previous script or back again.

9.3.9 Parse/Test single script

 Parse/Test single script

For a token or a production this function will parse the syntax of the actual script, to look for syntactical errors.

For a class element this action is disabled.

A single test will be executed for a test script.

9.3.10 Parse/Test all connected scripts

 Parse/Test all connected scripts

For a token this button has no other meaning, than to parse a single token, because tokens don't depend on each other.

In the production window all scripts will be parsed, on which the current script depends. This causes an exhaustive check of all these scripts. Following points will be tested:

Syntactical correctness of the scripts
Completeness of the scripts
Syntactical correctness of the interpreter code
Type check of the interpreter code

For a class element this action is disabled.

In the test window all scripts will be executed, which belong to the same group as the current script.

9.3.11 Parse/Test all scripts

 Parse/Test all scripts

On the **token and the production page** all scripts will be parsed, checked for syntactical correctness and completeness and the interpreter code will be checked also.

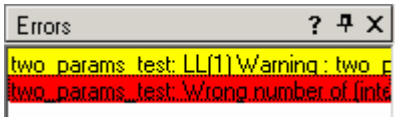
On the **production page** this function has the special effect that interfaces for all productions are created. So an interactive change of the current production while transforming one input text is possible.

On the **element page** all function scripts, variable declarations and variable initializations will be checked.

On the **test page** all tests are executed. Tests of a common group are parsed in common.

9.3.12 Error messages

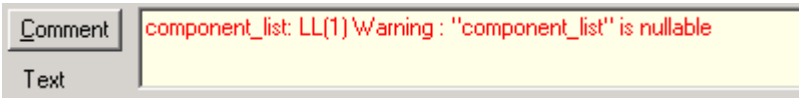
Error messages and warnings, which occur while parsing scripts, will be displayed in the *Errors* window*.*



The items of the list are colored:

yellow items are warnings
red items are error messages

After a single click with the left mouse button on one of these items, the whole message is displayed inside of the message window in the center of the user interface. By **F1 help** to the kind of error can be obtained.



After a **double click** on the line, the position in the script will be displayed, which caused the message

9.3.13 Clear semantic code

By the function: clear semantic code, all semantic actions can be removed from a script.
Parameters and return types are removed too.

On the token page in all scripts the parameters, return types and actions are removed.
On the production page, you can call this function either for an individual script or for all scripts of this page.
On the element page, there is nothing but semantical code. So you have the possibility to remove all scripts totally from the project.
On the test page, there is the possibility too, to remove all scripts totally from the project.

in the general Edit menu, there is the possibility to remove the semantic code of the token page, the production page and the element page at once.

Example:

If this function is applied ont:

```
Parameter: str& xs
return type:
Text:
  {{
  m_bInNewLine = true;
  node n("ClearItpText");
  }}
  Expression[n]
  {{
  xs = m_ftClear.visit(n);
  }}
  {-
  out << xs;
  -}
```

one gets:

Parameter:
Rückgabetyp:
Text:
  Expression

9.3.14 Import

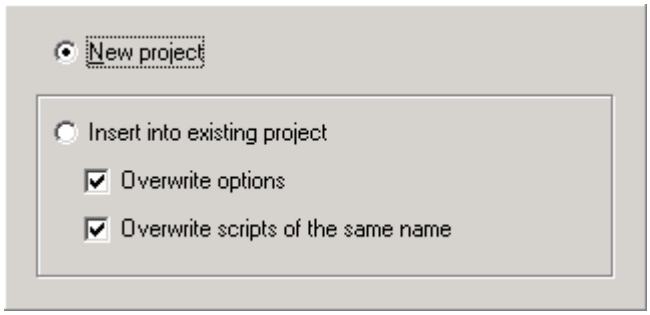
The menu item *Import* lets you import lists of scripts, which were exported in a previous session.

Depending on the actual page the export files will have different formats and extensions:

Type of script	extension
All	tti
Tokens	ttx
Productions	ttr
Interpreter	tte
Tests	ttt

It isn't checked, whether the imported productions, tokens etc. are written in a **correct TETRA syntax**. On the contrary: the syntax for the import is designed as tolerant as possible. Most syntax elements are optional. So the import is made easier of grammars which were written with other parser-generators. It is a disproportionately big effort to write exact conversion programs for such grammars. The possibilities of the IDE simplify the adaptation to the TETRA syntax enormously.

The choice exists at the import to make a new project or to insert the imported scripts in the project already existing, if a project is already opened in the TextTransformer.



The small box "**Overwrite options**" determines, whether the project options are taken from the import file or not. If the option is activated, the current project options then are overwritten, even if no options are listed in the import file. The default options are then set.

If the option "**Overwrite script of the same name**" is **not** activated, only those scripts are added to the repository, which have names, that differ from the names of scripts, that already exists in the repository. Scripts will not be overwritten by the import then.

The import format for a complete project has the structure:

```
ImExport ::=
             "TextTransformer"
             ProjectOptions

             (
               Tokens
             | Productions
```

```
              | Members
              | Tests
              )*
```

The formats for the export of the tokens, productions, interpreter scripts and the tests are identically with those of the respective productions of the complete format.

```
Tokens ::= "TOKENS" Token*
Productions ::= "PRODUCTIONS" Production*
Members ::= "MEMBERS" Member*
Tests ::= "TESTS" Test*
```

The details can be taken from the enclosed ImExport project. The structure of a production is given as an exemplary example here:

```
Production ::=
               Comment?
               IDENT
               Params?
               ReturnType?
               "::="
               LocalOptions?
               Field
```

Example:

```
/*
*/
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
)*
"<)"
<)
```

The definition of a production starts with its name. On this parameters in brackets follow optionally. A return value can again optionally follow behind a colon. The text of the production is included in the brackets "(>" and "<)". These tokens are chosen so that they might not appear within the text, The syntax for the other scripts is analogous. The number of the following texts corresponds to the number of respective entry fields is in the TextTransformer.

The mentioning of the **options** is also optional. For not mentioned options the respective default value is set.

9.3.15 Export

By means of the export item in the project menu the scripts are written into an ASCII-file.

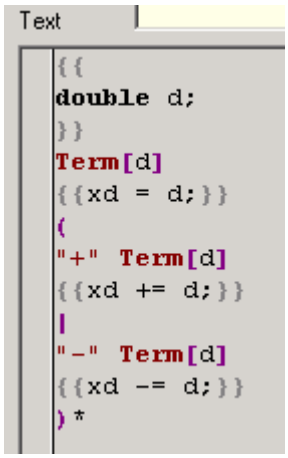
The exported files can be used as backup or for a reimport into other projects.

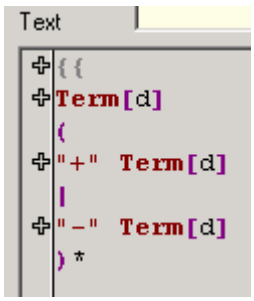
9.3.16 Collapsing semantic code

On the production page there is a special button:

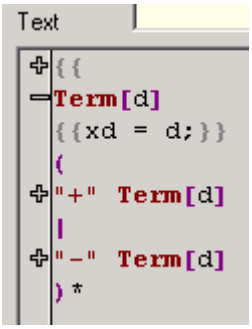
 Collapse semantic code

If this button is pressed, only the syntactical part of the script is shown and its structure is much more clear.
Normally the semantic code is characterized by inclusion into one of the pairs of brackets: {{...}}, {_..._}, {-...-} or {=...=}, as is shown in the left picture.
If this code is collapsed, these brackets are replaced by a plus symbol at the left border of the edit field, as shown in the right picture.





The plus symbol is shown in the line above the line where the opening bracket had been. This is not possible for the first action of the example, because it begins in the first line. Because of this the collapsing is incomplete. This is the case also, if one action is in the same line as syntactical code. By a mouse click on a plus symbol, the code of the corresponding action is shown again:



9.4 Debugging and executing

The productions and tokens can directly be tested and executed inside of the TextTransformer. To do this, first write or load a source text, which shall be analyzed or transformed. Now you have to choose a start rule from the combo box in the tool bar.

The program (the start rule) can be executed in different modes: step by step or at a stretch. In both case at first the start rule will be parsed and presented in the syntax tree together with its sub-rules.

There are numerous aids simplifying the **debugging** of TextTransformer projects:

- The recognized and expected text sections are highlightedly
- Breakpoints can be put in the text
- Breakpoints also can be put in the syntax tree
- The contents of variables can be looked up in the variable inspector
- You can jump back to the current position
- The information retrieved last is shown in the info-box
- Debug information is output in the log window

Further aids are put in the **navigation block**

- The recognized and the expected tokens are shown
- A stack of the called productions is shown
- The first sets of productions and branches are shown

9.4.1 Source text

Source text window:

You can select between a pure text viewer and an editor on the upper edge of the source text window. The text represented is the source text which is analyzed and transformed.

Editor (windows linebreaks)

To the start of the TextTransformers, at first the editor is active. So it is possibly to enter a text in the window or to copy a text into the editor. You can test the parser with this text and arbitrary variations of the text

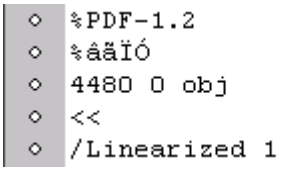
Viewer (read only)

Usually a source text available on the hard disk will be parsed. If such a file is opened it always appears in the viewer. The viewer shows the text exactly as it is stored on the hard disk. Also text files with line breaks not in conformity with Windows and even binary files are shown correctly and can be parsed. You, however, can't change the text.

The viewer can show the loaded file on three modes:

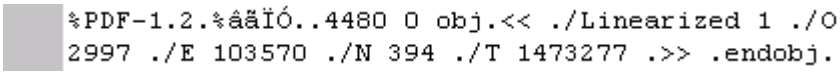
Text

In the text mode the line breaks are represented as usual.



Binary

In the binary mode line breaks and other control characters are represented by points.



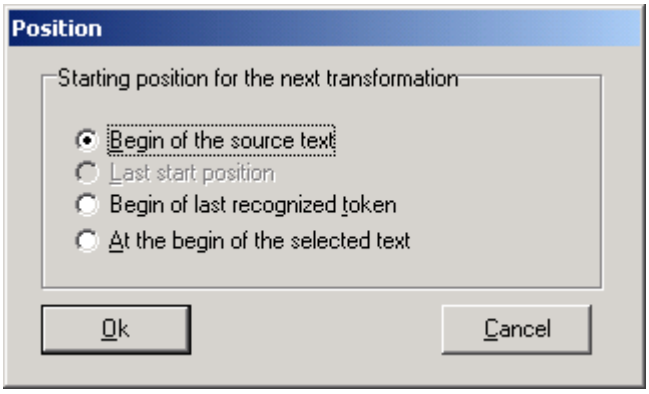
Hexadecimal

In the hexadecimal mode the hexadecimal values of the characters are shown together with them.

```
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 Section of text

The section of text, which will be transformed, depends on the actual state of the program and on the choice of the user. In the options for the user interface can be set, that a transformation shall begin always at the beginning of the input text, or if other possibilities shall be allowed too. In the the latter case the following dialog appears before the start of the debugger:



In the editor the option **At the beginning of the selected text** also is offered, if no text is highlighted. In this case the debugger starts at the current position of the cursor.

9.4.3 Enabling actions

The c++-interpreter can be disabled, to test only the analysis of the input. If however the text shall be transformed, this code must be executed. So you have to enable the actions.

Enabling and disabling the semantic actions can be done either by the menu: *Start->Action*, or by the checkbox in the toolbar.

actions enabled

actions disabled

If the parser uses semantic actions in IF-structures or WHILE-structures, the actions must be left enabled, because the project would not compile without them.

9.4.4 Choosing a start rule

To transform a text, you first have to choose a start rule. This choice is done in the according combo box of the tool bar:

Per default the production, which is set in the project options, will be selected in the combo box of the tool bar. If there is no explicit start rule set in the options, the name of the production will be displayed, which also is the name of the project. If there is no such rule, the selection remains empty and you have to choose a production manually.

Even productions with parameters, can be chosen as start rule. When such a rule is executed, default values are used for the parameters.

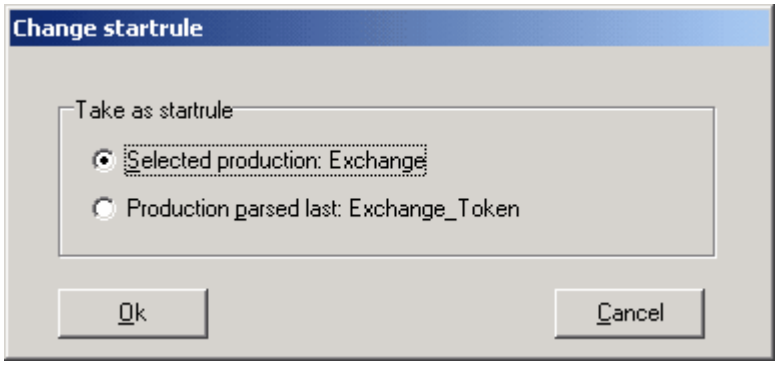
9.4.5 Interactive change of a start rule

If sections of text shall be transformed interactively by different productions, the according production can be selected in the combo box of the tool bar.
If the productions were compiled by Parse all scripts, an interactive change is possible every time. However, if the were compiled by Parse connected scripts (this is the case also, if you have started the transformation immediately on the main page), a just selected rule first must be (automatically) parsed eventually.
What exactly happens, if you change the start rule depends of the things done before.

9.4.6 Change of the start rule

What exactly happens, if you change a start rule depends of the things done before. During a TextTransformer session several situations can occur.

1. **Change of the page**: On the production page at last a different production was parsed (compiled) than is selected in the box of the tool bar. If you now go to the main page and start a transformation, the following box will appear:



2. **Change of the start rule by the box of the tool bar**: If the productions were compiled by Parse all scripts, a change is possible immediately.
However, if Parse connected scripts compiled them, the box above will appear, to confirm the change.

As well under point 1 as under point two there are two possibilities (if the scripts were parsed by Parse connected scripts):

a) **The new rule already is compiled:** If the new production is contained in the set of productions, of which the first start rule depends, the new rule will be parsed already too. But to execute it immediately is possible only, if the Interface option is enabled for this rule. If not, this rule first has to be compiled (automatically), to create a special scanner, which can test, whether the actual text is matched by one of the token of the rules first set.

b) **The new rule is not compiled already:** If the new production is not contained in the set of productions, of which the first start rule depends, it must be compiled.

9.4.7 Parse start rule

The selected production of the box in the toolbar can be parsed directly by means of the button (of the same toolbar)

If the actual page is not the Tetra page, the page will change to the production page.

9.4.8 Syntax tree

When a start rule was parsed, it will be displayed together with its sub-rules in a syntax tree. By a pop-up menu breakpoints can be set at tree nodes or accompanying first sets can be shown.

At first the productions appear in collapsed form.

Collapsed form

In the collapsed form only the names of the productions are shown, precede by a plus symbol '+'. (If the productions aren't parsed, the plus '+' is absent.)

Expanded Form

In expanded form the names are preceded by a minus '-' and below of the name the sequence of sub-structures is displayed.

Node icons

Each node of the tree is preceded by a little icon, which characterizes the type of the node. The names of the nodes are preceded by a prefix, which also denotes the type.

Icon	Type	Prefix
	Production	
	with local options	
	Call of a production	NT
	Terminal symbol = Token	T
	ANY-Symbol	ANY
	SKIP symbol	SKIP
	Alternative	Alt
?	Option	Opt
+	Repetition	Rep
	optional Repetition	OptRep
( )	counted Repetition	Count
IF	IF structure	If
W	WHILE structure	Cond
	BREAK	Br
	Semantic action	Sem

Concatenation

Parts of a production, which follow each other, will be displayed one beneath the other, conjunct by a vertical line with little arrows. A production

A = a b c d

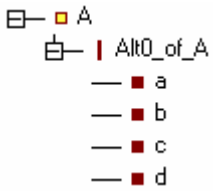
will be displayed as follows

Alternatives

For parts of a rule, which are alternative, an extra node is inserted, and beneath this node the alternatives are displayed as discrete nodes. A production:

A = a | b | c | d

will be displayed as follows



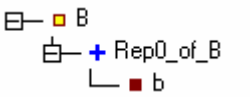
The name of the node, which combines the alternatives is constructed by the prefix "Alt_", a counter and "_of_", followed by the name of the superior node. From the name you can reconstruct the type and position of a node inside of the whole grammar.

Options and repeats

For options or repeats an extra node is constructed. The child node of the latter is that, what is optional or will be repeated. The production:

B = (b)+

is displayed as follows



The name of an option node or a repeat node is constructed analogous to the name of an alternative

node. The following expressions are used:

Opt	for options
0ptRep	for optional repeats
Rep	for repeats

Semantic actions

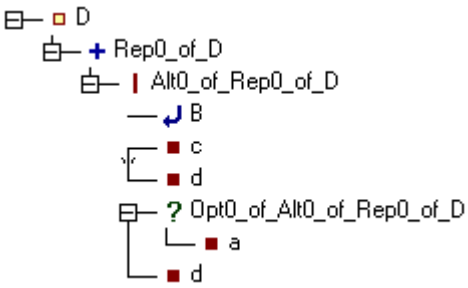
At positions, where semantic actions are executed inside of a production, in the syntax tree simple nodes are displayed. The names of the nodes are constructed from "_Sem", a counter and the name of the superior production. Actions, which are directly combined with a token are not displayed.

Complex example

If one of a group of alternatives consists itself of a sequence of token and productions, this sequence is displayed as a sequence of connected nodes. The whole sequence of this alternative is separated from the other alternatives.
Example:

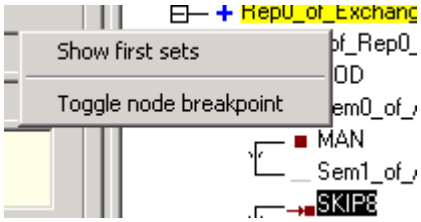
D = (B | c d | (a)? d)+

would be displayed in the syntax tree as:



9.4.8.1 Pop up menu

For each node in the syntax tree a pop up menu is shown, if you click on the node with the right mouse button.



You can get information about the context of a node, by:

Show first sets

You can set or remove a breakpoint, by

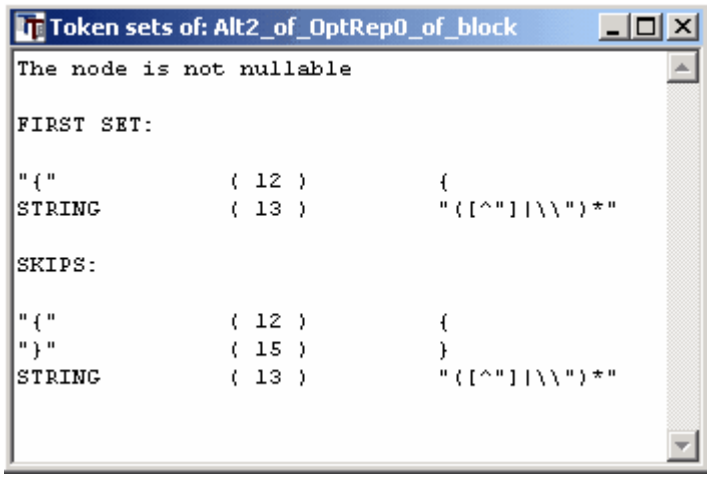
Toggle node breakpoint

The use of these items only makes sense, if the node is compiled.

9.4.8.2 Show first sets

By means of the pop up menu, that appears if the right mouse button is pressed for a node in the syntax tree, information about the first set of the node is displayed (if the node is parsed).

Example:

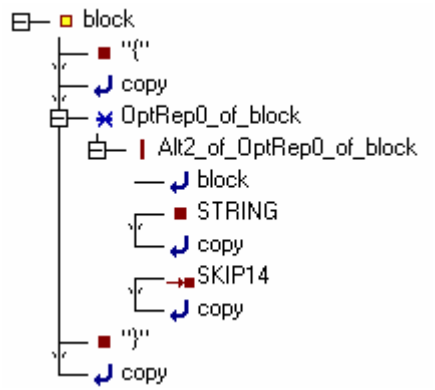


Remark:
Beginning with the version 0.9.8.8 in addition a line is displayed with the options valid for the node.
E.g. `Options: sep !icase, global ig lit (!all) !rgx`
The meaning of these abbreviations is explained below.

This box belongs to the alternative in the *block production* of the Guard example:

```
block =
          "{" copy_text
          (
            block
          | STRING copy_text
          | SKIP copy_text
          )*
          "}" copy_text
```

In the tree view:



1. Caption

The caption of the dialog displays the name of the node.

2. Header

In the first line is written, whether the node is nullable or not.

3. FIRST SET:

The lines of the *FIRST SET* list are consisting of the name, definition and symbol number of the first set of the node.

```
"{": { (12)
STRING: "([^"]|\\")*" (13)
```

All tokens are listed, which are beginning the alternative chains.

In the example the node presents the following alternative:

```
  block
| STRING copy_text
| SKIP copy_text
```

"{" is the single token, by which the *block-production can begin and* STRING is an alternative terminal symbol.

4. SKIPS:

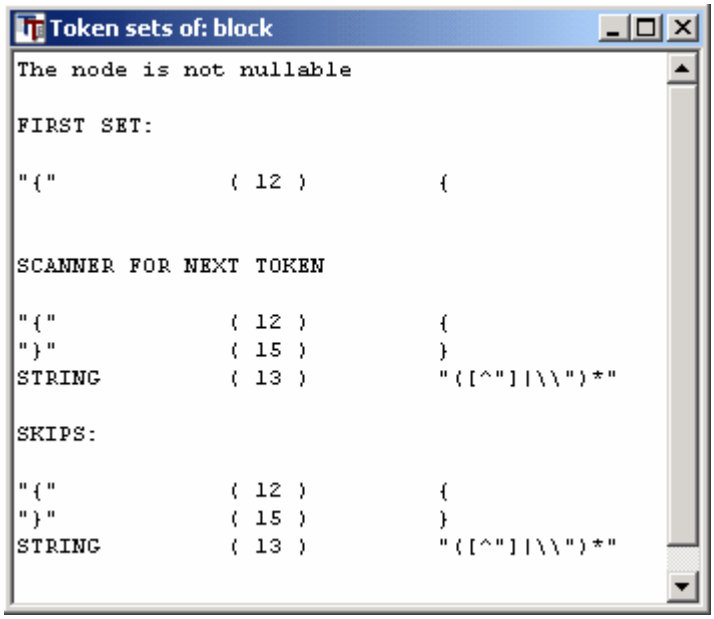
If a SKIP-symbol belongs to the alternatives of the node, or if the node itself represents a SKIP-symbol, the set of token, to which can be skipped, is shown here.
In the example:

```
"{": { (12)
"}": } (15)
STRING: "([^"]|\\")*" (13)
```

The next position, where one of these tokens appear in the input, will be searched, if none of the tokens, which are listed in point 3, are found at the actual text position

5. SCANNER FOR NEXT TOKEN

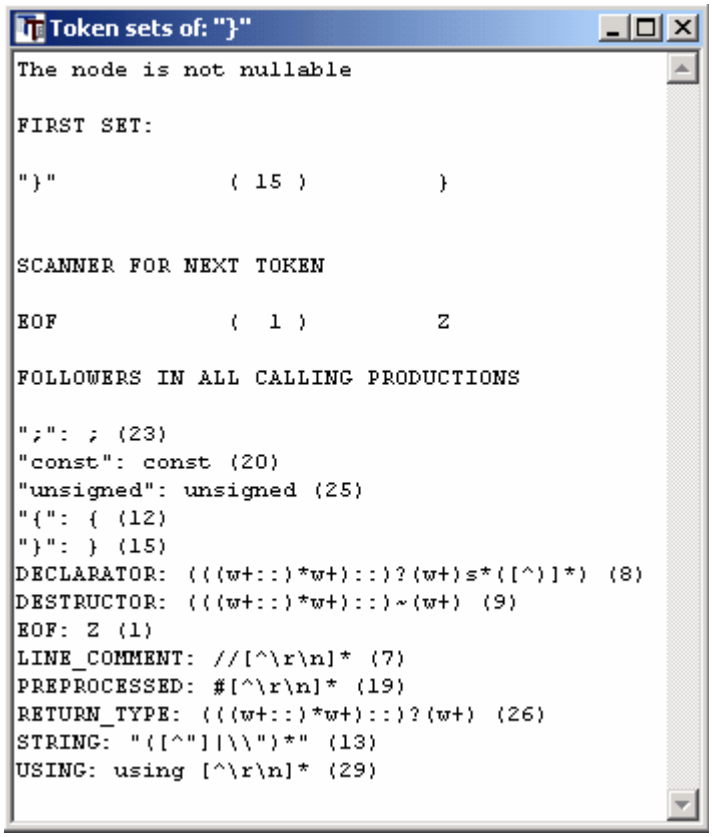
Only, if the node represents a terminal symbol or the call of a non-terminal, further token lists can follow for the "SCANNER FOR NEXT TOKEN". While in the lists at point 3 and 4 token are listed, which lead to the current node, in the following lists token are listed, which lead to the next node.
In the same *block*-production of the example above the display for the *block* node is:



After a recursive call of the *block*-production inside of itself, the next token must be found. The candidates for this search are listed in the lists following the title "SCANNER FOR NEXT TOKEN". Either with "{" a new block begins immediately or a string follows or the block will be left with "}". If none of these cases applies - because of the SKIP-symbol - the next position in the text will be searched, where one of these cases applies.

6. FOLLOWERS IN ALL CALLING PRODUCTIONS

Finally there is a different list of tokens, if the node represents a terminal symbol "at the end" of a production. "At the end" means, that there is an immediately follower of the node, which does not belong to the actual production.
In the *block* production, this applies to the node of the closing brace "}". The first sets for this node are the following:



In the list following the title "FOLLOWERS IN ALL CALLING PRODUCTIONS" all tokens are listed, which can follow the call of a production. Such a call is the call of the *block*-production inside of itself. Therefore followers of this call must be includes in the list of the followed of "}". In point 5 the list of the token, which can follow the call - the next token - already were listed.

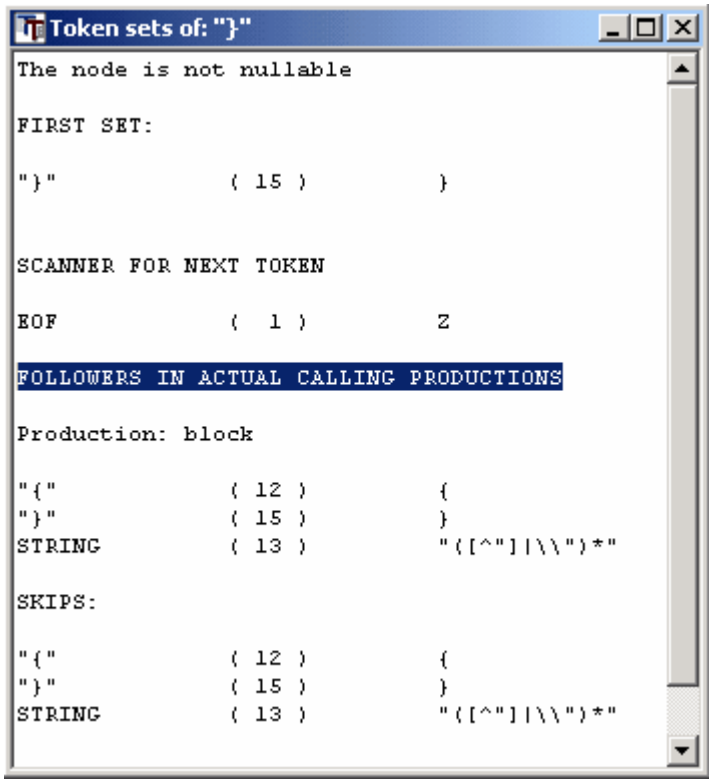
"{": { (12)
"}": } (15)
STRING: "([^"]|\\")*" (13)

The other "FOLLOWERS IN ALL CALLING PRODUCTIONS" are following the calls of *block* in other productions.

The last list is changed if it is shown while debugging the Guard-project. The list then will be renamed to:

7. FOLLOWERS IN ACTUAL CALLING PRODUCTION

Now only the tokens are listed, which at the actual moment really can follow, that means the followers of the actual call of the production. In the example: the followers of the call of *block* inside of the *block* production:



Now only the token discussed in point 5 are shown. Follower tokens of calls of block from other productions actually are irrelevant.

Options:

Beginning with the version 0.9.8.8 in addition a line is displayed with the options valid for the node.

Symbol	Abbreviation for	Meaning
!		not
sep	separated	word boundaries
icase	ignore case	ignore case
global		Settings for the global scanners
ig	ignore	ignorable characters
lit	literals	literal tokens
all		test all
rgx	regular expressions	not literal tokens

E.g.

```
Options: sep !icase, global ig  lit  (!all) !rgx
```

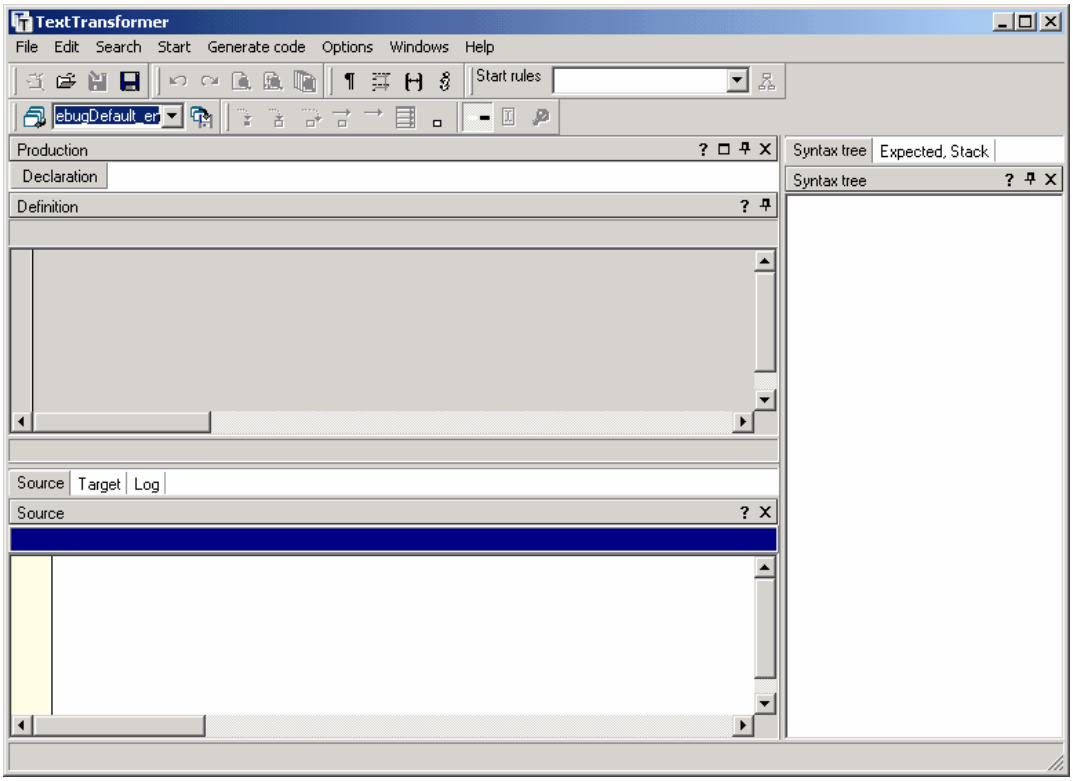
This means, that literal tokens must be separated words, that case is not ignored and that the global scanners for the ignored characters and for the literal tokens are used.

9.4.9 Start mode

There are different possibilities or modes, to execute a TETRA program. These possibilities are attainable either by the menu or by the buttons on the tool bar.

Next token (F6)
Step into (F7)
Whole routine (F8)
Start (F9)
Execute (F10)
Transformation of groups of files (F11)
Reset (Ctrl+F12)

If you chose one of the first five items, the application changes into the debug mode and the layout of the whole user interface is change too.



This is the "DebugDefault" layout. You can customize is to your own needs.

9.4.10 Execution step by step

The execution step by step is useful, to find positions, where errors (bugs) might occur.
You can execute the recognition or action of each node of the syntax tree. The node, that will be executed next is marked in yellow color.
At each step is tested, if the text, which follows the already parsed text, is matched by one of the token of the first set of the actual node. If this is not the case, the parsing has failed and will be stopped with an error message. If a matching token is found, the next node will be marked.

What is the next node depends on the kind of step you make.

Next token (F6)

The rules are executed to the next terminal node. There might be some branches and semantic actions performed until there. The actions are executed if the interpreter is enabled.

Back to the last token (Shift + F6)

You can go back virtually to the previous token by this button. Semantic actions aren't undone. When making progress once more no semantic actions are executed, until the position which was already achieved is exceeded.

Step into (F7)

The child nodes of a production, an option, a repeat or an alternative will be executed one by one. Depending on the position of the actual, yellow marked node, there are following possibilities:

a) If the actual node represents a branch (option, repeat or alternative), the debugger will step into the first child node.

b) If the actual node represents the call of a production, the debugger will jump to the representation of that production.

c) If the actual node represents a terminal symbol, that is a leaf in the tree, the execution will lead to the following node (below), if there is a node. Otherwise, the terminal node is the end of a chain and the step will lead to the node behind the parent of the terminal.

Single step back (Shift + F7)

You virtually can go back one step by this button. Semantic actions aren't undone. When making progress once more no semantic actions are executed, until the position which was already achieved is exceeded.

Step over (F8)

In this mode a whole branch is executed. The result is the same as a *Step into*, if the current node

is a terminal node or represents a semantic action. If the current node represents a whole structure, this is executed in one step.

9.4.11 Execute a look-ahead step-by-step

Productions can be executed on a trial basis. In dependence of the success the same production is executed or another branch is chosen. Such a look-ahead also can be tested step-by-step.
This is possible in arbitrary staggering. I.e. the success of a look-ahead can depend on further look-ahead's which are carried out within the first one.

Level of look-ahead

The level of the look-ahead is shown on a little field within the tool bar. An empty field or a zero means that the parser isn't within a look-ahead but in the main stream.

Into the look-ahead (Ctrl + F7)

If the parser is at the beginning of an IF or a WHILE structure like shown below,

```
WHILE(declaration())
 declaration[root]
END
```

you can step into the corresponding look-ahead by this button. At other positions the button works just like the button for a single step within a look-ahead level. If the expected token doesn't belong to the first set of the look-ahead-production, there is no change of the level too.

Different highlighting of the symbols shows, that

qualified_id it will be testet next

qualified_id it has been testet successfully

ptr_to_member the test failed.

Out of the look-ahead (Ctrl + F8)

You can leave a look-ahead by this button. All remaining steps within the current level are executed at once and the parser stops at the next higher level.

Remark: you can use the other buttons and functions of the debugger the same way within a level of a look-ahead like at the level of the main parser.

9.4.12 Execution at a stretch

If you execute a program at a stretch, the input will be parsed till its end or up to an error. This can be done in two modes

Start (F9)

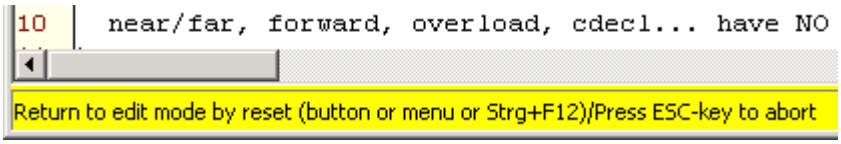
In this mode the execution will be stopped at break points and then can be continued step by step. If an error occurs, there will be detailed information about the circumstances.

Execute (F10)

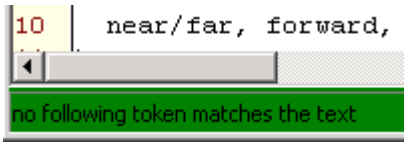
This mode is for the fast execution of ready programs. Some steps, which are made in the other mode, here are left out. But there will be given only general information, if an error occurs.

9.4.13 Checking success

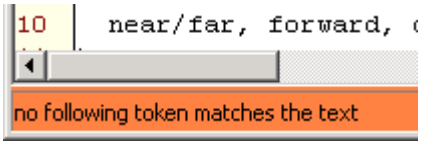
As soon as the debugger is started, the status bar is colored yellowly. The yellow color indicates that no fault has appeared till now, but the processing isn't completed yet.



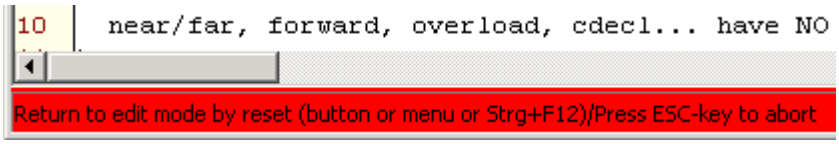
If the execution of the transformation was successful, then this is signaled by a green colored status bar in the debug mode.



If no following token is found before the text ends, the parser is stopped and the status bar is colored orangely.



Furthermore it is possible to execute single steps because semantic actions still can follow on the abortion of the recognitions.
A red status bar finally shows that the parser finished with a fault.



If the source text wasn't processed completely, the following lines are appended to the target text:

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XX                                                                  XX
 XX   An error occurred. The transformation is incomplete!           XX
 XX                                                                  XX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

More information about the fault is shown in the log window.

9.4.14 Reset

By

 **Reset**,

the execution mode of the TextTransformer is finished. Expanded branches of the syntax tree are collapsed and if you click on the name of a production, its definition on the production page is shown.

If an output was generated, the following dialog will ask you to delete it or not.

9.4.15 Mark recognized/expected token

If the program is executed step by step, not only the actual node in the syntax tree is marked, but also the actual position in the input text. You can choose between a selection of the last recognized token and the token, which is expected next. This choice is most comfortably carried out via a tool button. If the button is up

the last recognized part of text remains marked, until the next terminal node in the syntax tree has been passed.
If the button is pressed, then immediately after a token was recognized, the text section that corresponds to the expected next token is marked.

9.4.16 Breakpoints

You can stop the execution of a program at a definite position of the input or at a definite node in the syntax tree and test the further processing step by step.

Text breakpoint
Node breakpoint

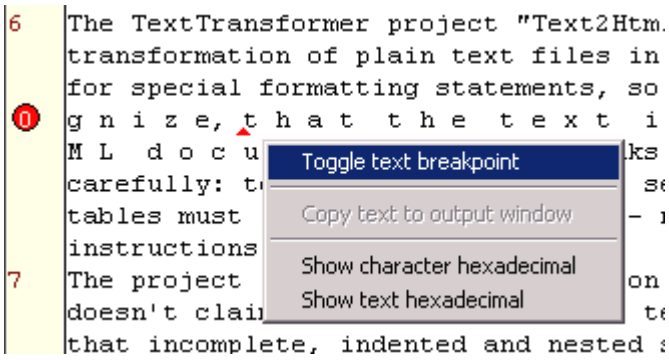
9.4.16.1 Text breakpoint

After the recognition of a certain position in the input text, the program can be stopped by a break point.

First you have to put the mouse cursor on the desired position in the input text. Now you can set the break point with the pop up menu, which appears, if you click the right mouse button. You also can use the menu: *Start->Toggle breakpoint*

Toggle breakpoint

Now in the same line the border of the source window is marked red. In the editor a red point with a white digit is displayed.



If you toggle the breakpoint again, the breakpoint will be removed.

A breakpoint at the beginning of a line also simply can be put and removed by a mouse click on the margin.

In the editor only:
You can set breakpoints with a special number by pressing Ctrl + Shift + Digit at the same time. With the same key combination a breakpoint can be removed if the cursor is in an arbitrary position of the line.

Altogether, ten breakpoints (0 - 9) can be set. You can jump to one of them in the editor by pressing Ctrl + Digit.

If the text is edited after a breakpoint was set, the breakpoints will be shifted. They should be cleared before editing the text.

The menu item in *Start*:

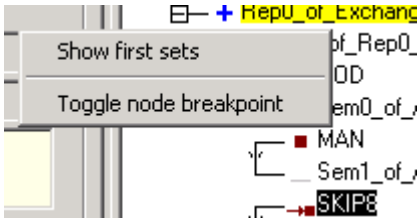
Clear text break points

removes all breakpoints in the text.

9.4.16.2 Node breakpoint

Before a node is executed, the program can be stopped by a node breakpoint.

You first have to select the according node in the syntax tree with the left mouse button and then to display a pop up menu by the right button. In the pop up menu the item *Toggle node breakpoint* sets the breakpoint.



A node with a breakpoint is displayed in red color. The same item of the pop up menu can remove breakpoint.

If the rules are parsed after a breakpoint has been set, all node breakpoints will be removed.

Node breakpoints can be put only in rules of the main parser. look-ahead productions and sub-parsers are interpreter calls. To put node breakpoints there, they must be put for a test as a start rule.

9.4.17 Recognized token

A docking window of its own is a little box with information about the recognized last token and the next token found.

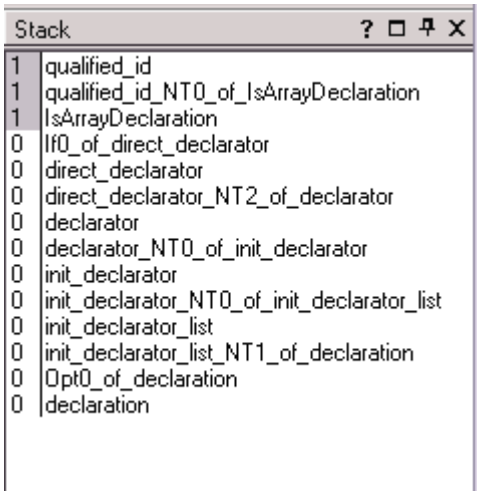
	Name	No
recognized		
found	"init"	14
found		

If a TETRA program is executed step-by-step, in this little table in the first row the current recognized token and in the second row the found next token are shown. As soon, as the node, which represents the found token is left, the found token is accepted and becomes the recognized token. At the same time the next token is evaluated.
Special cases are the SKIP-nodes. As soon as a SKIP-alternative was chosen, also the token, which follow the SKIP-node is known. It is shown in the third line of the table.

The combo box above of the token table contains a list of all tokens, on which the actual parsed productions are depending. For each of these tokens the name, symbol number and regular definition is displayed.

9.4.18 Stack window

Another window of its own is the stack window.



The stack window contains a list of all productions and branches, which are superior to the current node. By stepping from one production into a different or, if an optional branch is chosen, the name of the superior node is put into the first line of the stack window. The previous lines are moved one line below. So in the stack window, the sequence of nodes is displayed, which characterize the "way", to reach the actual node.

The preceding number denotes the level of look-ahead..

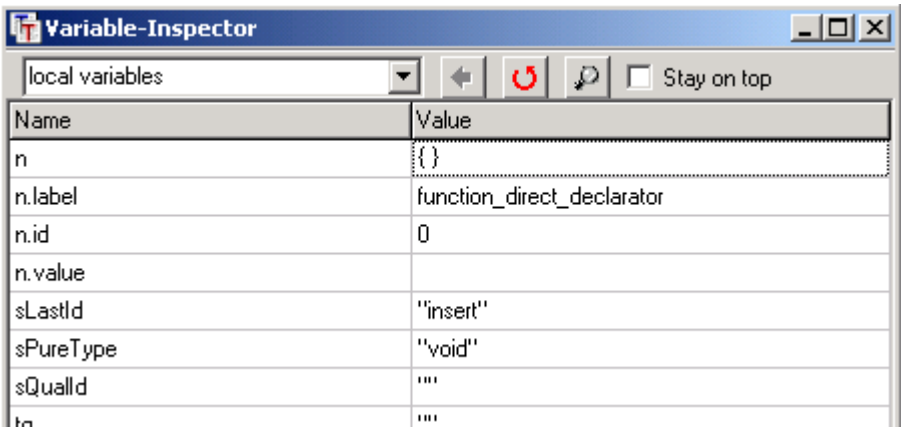
If you click on an item in one of the lists, the according node will be shown in the syntax tree and the section of text is marked too in the input window, which was recognized by the corresponding structure..

9.4.19 Variable-Inspector

By the variable-inspector you can look at the contents of the variables of the actual scope of a debugging session.

If you are in the debugging mode (not during a look-ahead, see below), you get the window of the variable-inspector by the button

or by the according item in the start menu



You can either write the name of a special variable into the field of the combo box at the left top of the dialog or select one of the five predefined items:

class variables

If you choose the item *class variables,* the source and target information are shown as well as all variables, which are defined on the element page. (The parser itself is denoted as *this* or (**this*).)

local variables

If you choose the item *local variables,* the values of all variables are shown, which are in the actual scope. These are the variables passed to the actual production and the variables, which are locally declared in the production.

xState (parser state)

If you choose the item *xState,* all elements of the parser state variable are shown, including all sub-expressions of the last recognized token.

Plugin Variables

If you choose the item *Plugin variables*, all variables of the plugin are shown, especially the source and target specification and the state of the indentation and scope stack.

DOM

If a DOMDocument has been created, it can be viewed here.
.

Buttons of the toolbar

Up in the hierarchy 

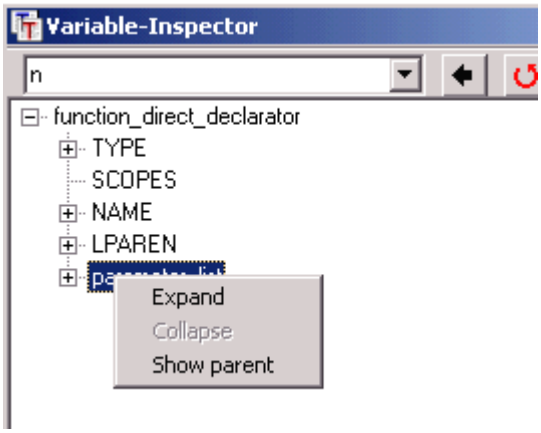
With the *Back*-button you can go a level higher in the hierarchy of the class elements or finally to a view of all variables of a visibility area (see above).

Actualize 

After you have opened the variable-inspector, you have to actualize the value of an eventually already chosen variable by the *actualize* button.

Details = 
Choice of a single variable by double click

If several variables are shown, you can select one of them with the left mouse button on the value side and view their content by the *Detail*-button or by a double click. So longer texts or the elements of containers or tree structures - like shown below - can be seen.



In a tree view nodes can be selected with the right mouse button. Then you can expand or collapse the whole branch, if you chose the corresponding items in the pop-up menu. You also can move the root of the tree to it's parent node.

Stay on top

If the check box *Stay on top* is actualized, the variable-inspector remains visible on top of the screen during the complete debugging-session. After each debugging step the content of the selected variable is actualized automatically. If *stay on top* is not set, there will be no such actualization and the inspector will be deleted from the screen by each step.
If the inspector is closed, while the check box is checked, it will be unchecked.

If it is a complex variable, as for example the xState-variable, the values of its class elements will be listed. In some cases some properties of the variables will be added to the list of values. As containers (mstrstr, vstr) can contain very much elements, for them only the number of elements is shown.

When debugging a look-ahead the variable inspector isn't shown because during a look-ahead no semantic actions are executed.

9.4.20 To the actual position

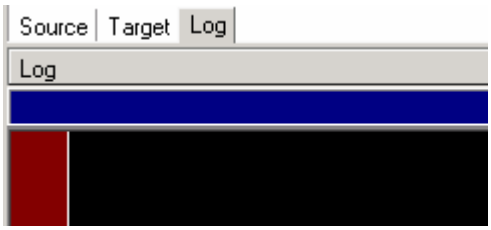
This function restores the state of the debugger after the last step: the last recognized token will be marked in the input and the last node in the syntax tree will be selected.

9.4.21 Info box

By the menu: *Start->Show last message* or the button

the last message, which was shown in the info dialog, will be shown again. This can be e.g. the last shown token sets.

9.4.22 Log window



Meta informations to the course of a program will be shown in the log window. These are messages about the success or about errors as well as the output, which is written explicitly to log by the programmer.

A small red box signals, that the window contains information..

9.5 Transformation of groups of files

There are two possibilities to transform files in batch mode:

1. interactively directly out of the TextTransformer or
2. by means of an additional command line tool: *tetra_cl*.

9.5.1 Transformation manager

The transformation manager is a dialog, by which you can transform whole directories or other groups of files.
You can reach the transformation manager either by the menu item *Transform groups of files* of the *Start* menu or by the according button in the tool bar:

Before the dialog opens the actual start rule - which will be used for the transformation - must be compiled. If this is not the case, it will be done automatically when you open the manager.

At first the button in the tool bar of the manager for executing the transformations is deactivated since no source files are selected for the transformation yet. Only if this has happened and options are set as requested, the transformation can be started. Before starting the transformations, you can check the list of the files which will be produced. There is a page of his own for each of these steps in the transformation manager:

1. Source files
2. Tranformation options
3. Preview of the list of target files
4. Results

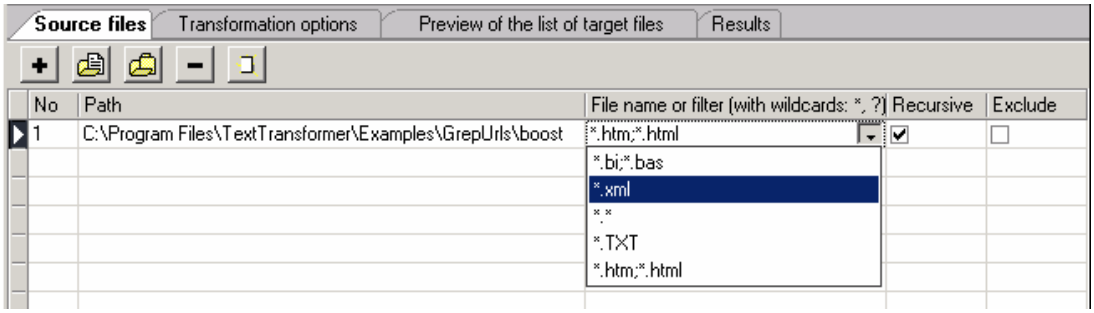
The settings, inclusive of the select folders and files, can be stored as a management and loaded when required newly.

9.5.1.1 Defining a new filter

The choice of the source files is made easier if, before, file filters are already defined for file types frequently required.
By the menu of the transformation manager ***New file filter*** a new mask for the files, which shall be transformed, can be defined. The dialog, which appears is the same, which is shown for the environment options.

9.5.1.2 Selecting source files

The files which shall be transformed are selected on the first page of the transformation manager and are shown in a table.



The page has a tool bar of its own with the buttons:

 Insert an empty row

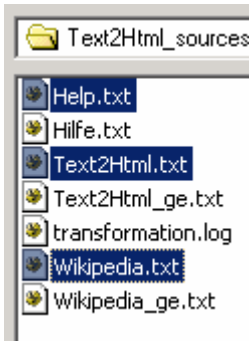
 Select a single source file

 Select a whole source directory

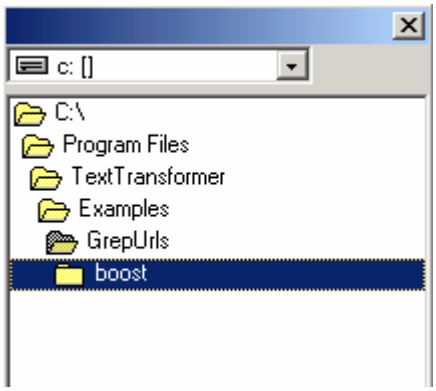
 Deleting a row

 Clear the whole table

The choice of a file or a folder is carried out respectively with a corresponding selection box. Several files also can be selected at once in the selection box.



After the confirmation of the choice a new row is inserted in the table below the tool bar for every file or every folder.



There are five columns in the table:

No

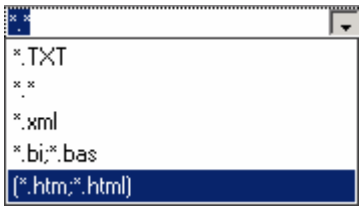
a simple counter

Path

The absolute path of the file or folder.

Filename or filter

For files the file name can be seen here (with extension).
For folders a filter can be specified here. Filters already defined can be selected from a combo box in this column by the mouse,



however, also arbitrary other filters can be written directly to the field. E.g. with the filter "*.txt" only the files of the folder will be transformed, which have the extension "txt".

Recursive

The check box in this field can be activated only for folders. If it is activated, then all files in the sub-folders of the shown directory are transformed too.

Exclude

Normally the check box of this field remains deactivated. However, it can be that you want to except some files or folders from the transformation of a folder. This is possible by producing rows of their own for these exceptions in the table and activating the excluding check box by mouse.

Extra

Extra parameters per file or file group can be put in the last field.

9.5.1.3 Transformation options

The Config button opens a small editor, in which you can write configuration parameters for the transformations.

In principle, there are two ways in the transformation manager to transform the source files:

1. **N:N**: each source file is mapped to a different target
2. **N:1**:. all files are mapped into a single target file

There are different sets of options depending on, whether an N:N or an N:1 transformation is intended

Log-file

Notwithstanding the way of the transformation, however, a log-file in which the actions and reports of the transformations are recorded can be determined. Such a log-file can contain more information than the messages reported on the results page.

9.5.1.3.1 N:N Transformation

N:N: each source file is mapped to a different target

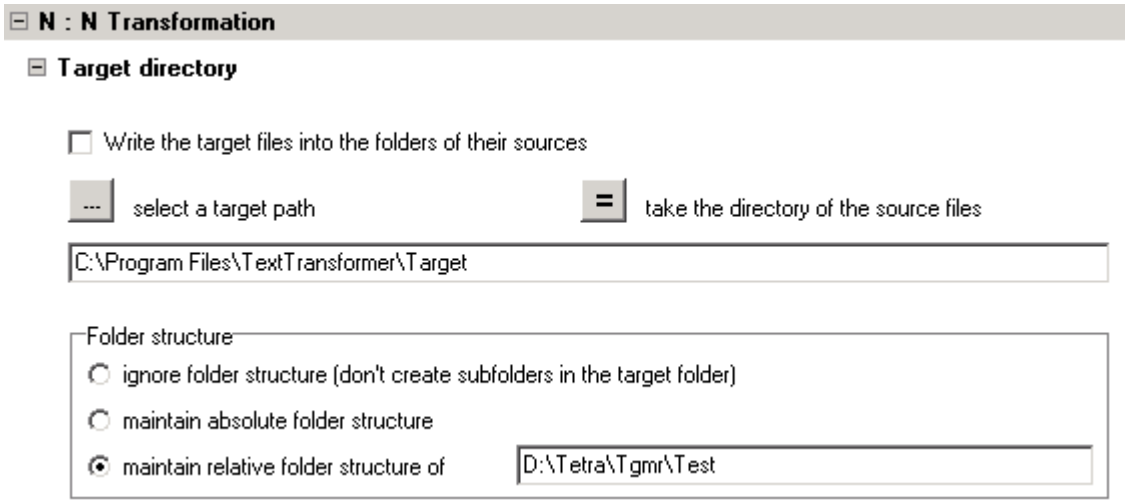
This is the standard case of the translation of a source text into a target text by the transformation manager. The same number of target files is created as the number of existing source files. In principle it is allowed to overwrite the source files by the targets. But you have to take care for an according saving. The TextTransformer optionally can make a backup into a chosen directory, before the transformation starts.

9.5.1.3.1.1 Select target directory

Both is possible: to overwrite the source files and to create texts in another folder.

Write the target files into the folders of their sources

The source files are **overwritten** when "Write the target files into the folders of their sources" is activated and no particular pattern for the target files is intended. Neither a separate target directory needs to be selected nor a folder structure fpr the targets has to be chosen. So the according fields are disabled.



Writing text into a specified folder

If the check box "Write the target files into the folders of their sources" is deactivated, the input fields for the target folders are enabled.

After you have just opened the transformation manager for the first time, the target directory is set to the directory, which is set in the environment options. But it can be changed temporarily.

By the button

a dialog for the selection of a different target directory is opened.

The button:

can help to navigate faster to the new target directory

Folder structure

If all source files have the same path, the following options for the folder structure don't matter. If they are, however, from different directories, then there are several possibilities for the construction of the paths of the target files:

Ignore folder structure

means, that all target files get the same path of the target folder.

Example:

If the target directory is: C:\targets

the transformation of the files

C:\program files\TextTransformer\source.cpp
C:\program files\TextTransformer\Sources\source.txt

results in the following target files:

C:\targets\source.cpp
C:\targets\source.txt

If there would be the source file "C:\source.txt" too, two of the resulting files would be the same. In this case an according error message will be produced.

Maintain absolute folder structure

Example:

If the target directory is: C:\Targets

the transformation of the files

C:\program files\TextTransformer\Source.cpp
C:\program files\TextTransformer\Sources\Source.txt
C:\Source.txt

results in the following target files:

C:\Targets\program files\TextTransformer\Source.cpp
C:\Targets\program files\TextTransformer\Sources\Source.txt
C:\Targets\Source.txt

The file "C:\Source.txt" makes no problems here.

Maintain relative folder structure

Example:

If the target directory is: C:\Targets

the transformation of the files

C:\program_files\TextTransformer\Source.cpp
C:\program_files\TextTransformer\Sources\Source.txt

results in the following target files:

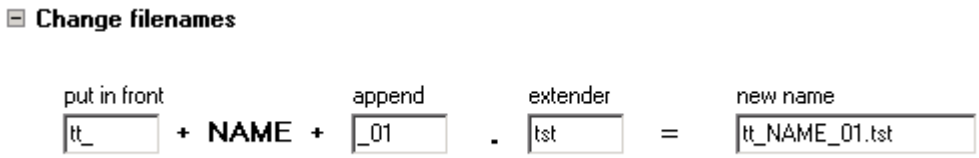
C:\Targets\Source.cpp
C:\Targets\Sources\Source.txt

If there would be the source file "C:\Source.txt" too, the starting directory for the relative folder structure had to be moved and one gets the same result as in the case of the keeping the absolute folder structure.

9.5.1.3.1.2 Setting pattern for the target files

The names of the source files can be changed during the transformation in the transformation manager. They can be provided with a prefix, a postfix or a new extension. The pattern for the names of the target files is determined by the according fields.

Example:

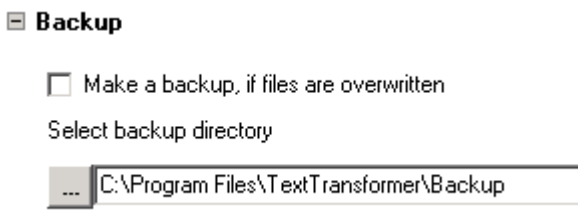


changes the filenames in the following way

test.dat	->	tt_test_01.tst
Source.txt	->	tt_Source_01.tst

9.5.1.3.1.3 Backup

If by the transformation existing files are overwritten, then it is advisably to make a backup of them before. There therefore is the possibility in the options of the transformation manager of selecting a folder into which the original files are copied before the transformation.



Even if the backup option is activated, a backup is made only, if at least one file will actually be

overwritten. In this case all source files are saved.

Before a transformation starts, it is checked whether, source files will be overwritten. If this is the case and the backup option isn't set, a warning appears which still permits to let make a backup of the the original texts.

By the roll back function the backup files can be copied back, as long as the settings weren't changed in the transformation manager.

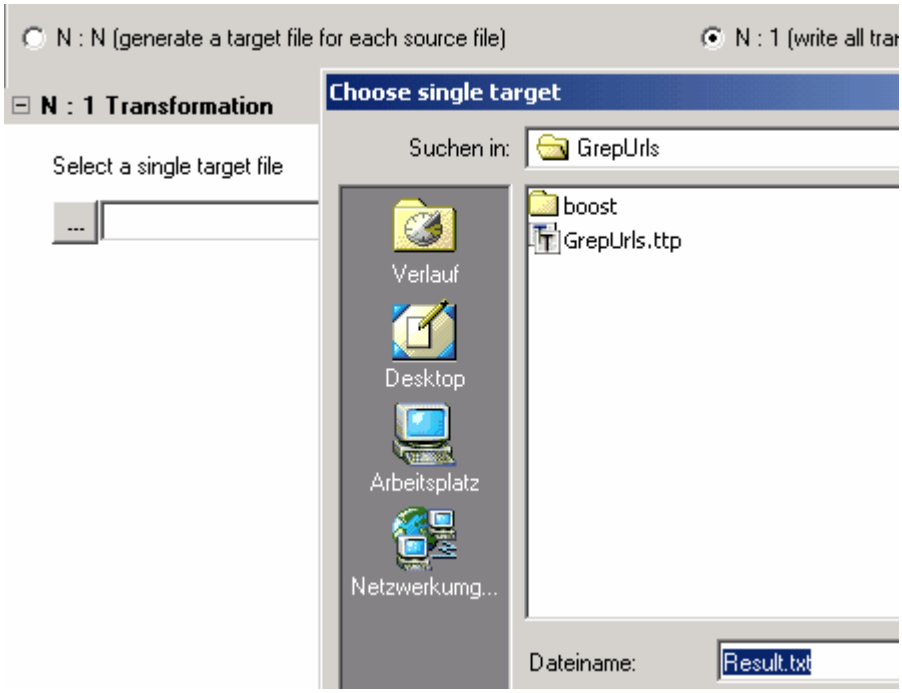
Remark: If the path and the name of a target file are identical with those of the source file, the result of the transformation will first be written into a **temporary file**, which will be renamed to the original name, if the transformation succeeds. If an error occurs, the temporary file will not be renamed and is left in the target directory. You can look at it in an editor and see where the error occurred. The names of the temporary files are constructed of *temporary*, eventually followed by a number and with the extension *tmp*.

9.5.1.3.2 N:1: Transformation

N:1:. all files are mapped into a single target file

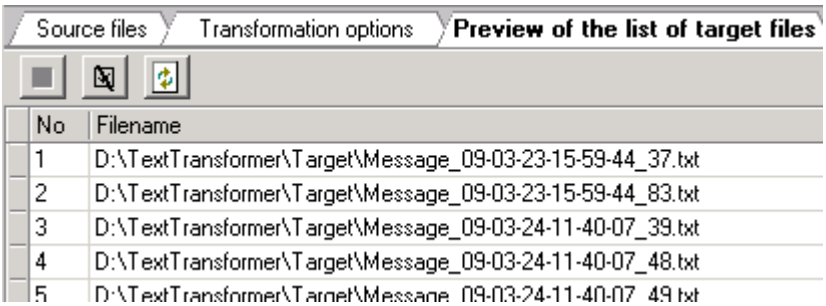
If information out of a multitude of files shall be extracted by the transformation manager into a **single** different target file, there is a N:1 relationship between source and target. In this case a target file has to be chosen, which is different to all source files. So a backup of the source files is not necessary.

The target file can be opened by a file select dialog. You also can use the dialog for the choice of the directory and enter the name of a file not existing yet in the dialog.



9.5.1.4 Preview of the target files

The list of the files which will be produced are shown on the third tab-page of the transformation manager.



In case of an N:1 transformation the different rows of a table show which source files can contribute to the file to be produced.

Excluding individual files

Similar to the table of the source files the table of the expected target files contains a field with a check box to exclude individual files. Here you can exclude single files whose source files are in a selected source folder.

Remark:
When writing a management the source files belonging to the excluded target files are excluded. If the management then is loaded, these files appear as excluded in the source files table and no more in the target files table.

Excluding successful transformed files

If, currently, the transformation of the files being part of the management was executed already once, then by the button all those files can be excluded from another transformation which were already transformed successfully. So it is possible to apply a corrected project alone to the files which, till now, couldn't be transformed.

Including all files

By the button all selections for the exclusion of files can be removed.

Actualize

You can refresh the list of files by the button .

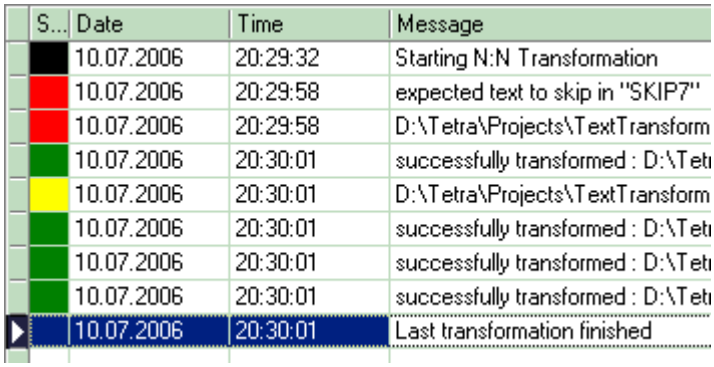
9.5.1.5 Start the transformation

The transformation of the selected files in the transformation manager is started by the menu item *Start transformation* or by the button in the main tool bar

When the transformations are started, the page is changed to the Results-page automatically.

9.5.1.6 Results

The rows of the table on the result page of the transformation manager contain messages which arise during the transformation of files. Messages are produced by the transformation manager automatically. However, messages, programmed from the user are shown too.
Every message is immediately written into a new row of the table after the message was created. So, the growing row number of the table at the same time shows the progress of the transformations.



S...	Date	Time	Message
	10.07.2006	20:29:32	Starting N:N Transformation
	10.07.2006	20:29:58	expected text to skip in "SKIP7"
	10.07.2006	20:29:58	D:\Tetra\Projects\TextTransform
	10.07.2006	20:30:01	successfully transformed : D:\Tet
	10.07.2006	20:30:01	D:\Tetra\Projects\TextTransform
	10.07.2006	20:30:01	successfully transformed : D:\Tet
	10.07.2006	20:30:01	successfully transformed : D:\Tet
	10.07.2006	20:30:01	successfully transformed : D:\Tet
	10.07.2006	20:30:01	Last transformation finished

In the first row the status of the message is shown as a color.

Color	Status	user defined by
	new source file	
	neutral information	AddMessage
	success message	–
	warning	AddWarning
	error message	AddError

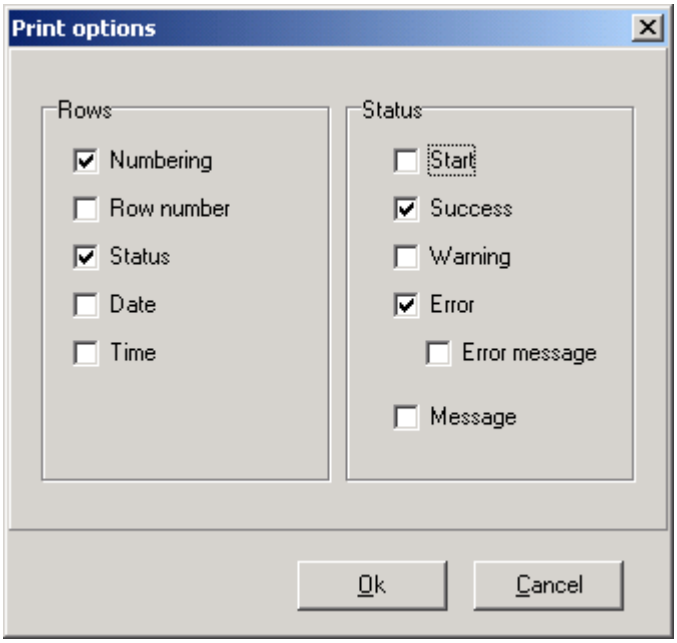
A report can be made from the list of results.

9.5.1.6.1 Report

A report can be written from the list of the results in the transformation manager. This can be done either by the menu item "File -> Save results as ..." or by the button on the result page

Now you either have to select an existing file or you can choose a directory and write the name for a new file into the according field.

The information which is included in the report can be determined by the user. Only the information will be written for which the according boxes are checked.



A numbering of the source files is recommended because sometimes there are several messages to one file.
The different color values of the first column of the result table are corresponding to the status boxes here. Only those lines are written, which have a selected status.

9.5.1.7 Corrections

After the transformation has finished, you can examine the transformed files and if needed, make corrections on the project.
To **look up a single transformed file** or to debug a source file, which produced errors, you can choose the according line in the table. By a **double click** on the row in the table or by the popup menu, the dialog will be closed and the source file will be loaded into the input window of the IDE. If the source file was transformed successfully, the target file will be loaded into the target window too.

You can improve the project and reopen the transformation manager. The settings are the same as before, if you have not opened another project meanwhile.So you can repeat all transformations with your improved project.

9.5.1.8 Roll back

If the project was improved, you can repeat the transformation of the same source files to the same target files. You can copy back the backup files before by *Roll back,* if the original files were overwritten, .

Attention! If the source files were overwritten, and no roll back was made before a second transformation, the backup himself will be overwritten by the results of the first transformation. **It is always more save to choose a target directory different from the source directory.**

9.5.1.9 Management

The sum of the settings of the transformation manager is called a management here. A management is a transformation manager project. To prevent a mistake with a TextTransformer project, it is described as a management.

By the menu item: ***Save management as***. you can save a management

By the menu item: ***Open management***, you then can reload a management. Recent managements can be accessed quickly by the history list in the menu.

A management can be used to control the commandline tool Tetra_cl too.

Managements are save with the extension "ttm". They are parsed by the project FileList.ttp.

The syntax for a management was designed as scarce and simple as possible, so that it also can be written by hand. A management consists in the extreme case in only one file path.
Fields to inactive options aren't stored in a management also if they are readable in the transformation manager. When parsing a management, values are deciding by their existence, whether the options for which they are needed, are active or not. E.g. the existence of a target path decides, whether files are written over or not.

An example management is in the folder of the GrepUrls example:

```
single_target = C:\Program Files\TextTransformer\Examples\GrepUrls\Result.txt
log_file = C:\Program Files\TextTransformer\Log\transformation.log
+ r C:\Program Files\TextTransformer\Examples\GrepUrls\boost\*.htm;*.html
```

9.5.2 Command line tool

A command line version of the TextTransformer exists: **tetra_cl.exe**. By means of this program an automatic transformation of groups of files is possible. Since version 1.2.2 the command line tool can optionally be installed with the installation of the TextTransformer. It then is copied into the bin folder of the program directory.

When calling **tetra_cl**, you have to pass some parameters concerning the files, which shall be transformed and how.

9.5.2.1 Parameter

A call of **tetra_cl** has the following form:
Tetra_cl can be controled either by a management, which was produced with the transformation manager or by parameters for the source and target files.
In the first case a call has the form:

tetra_cl -p PROJECT -m MANAGEMENT [-a]

and in the second case:

tetra_cl -p PROJECT -s SOURCE [-t TARGET] [-b BACKUP] [-c CONFIGURE] [-x EXTRA] [-a] [-r]

Parameter are specified by single letters preceded by a dash '-' and sometimes followed by a space and some text. Expressions in brackets are optional.
If a path contains spaces, it has to be quoted.

Parameter	Meaning	Examples
-p PROJECT	TETRA project	exchange.ttp
-m MANAGEMENT	a project file made with the transformation-manager	MyWebSite.ttm
-s SOURCE	Source file(s)	C:\dir*.txt
-t TARGET	Target file or directory	C:\dir2\target.txt
-b BACKUP	Backup directory	C:\Backup
-c CONFIGURE	Configuration parameter	"\"C:\\boost\", \"C:\\mylib\""
-x EXTRA	Extra-parameter	alternative_rule
-a ASK	Ask at each file	
-r RECURSIVE	recursively including the files of the subfolders	

-p PROJECT

The parameter -p must be followed by the address of the TextTransformer project, by which the files of the source directory shall be transformed.

-m MANAGEMENT

The parameter -m is followed by the address of the transformation manager project, which specifies the source and target files.
If an -m paramerter is provided, -s, -t and -r are ignored.

-s SOURCE

The parameter -s must be followed by a specification of the files, which shall be transformed.
In the simples case this a specification is an address of a single file, like "C:\dir\source.txt". To transform all "txt" files of a directory, you can use a mask like: "C:\dir*.txt". If there is no directory specified in the mask, all files of the actually directory will be transformed; e.g.: "ab?.*" will chose all files of the actually directory beginning with "ab" followed by a single character and an arbitrary extension, e.g. "ab1.txt", "ab2.txt" and "ab_.bat"

-t TARGET

The specification of a target is optional. If there is no, all source files will be overwritten by their transformed versions. You can specify a fully qualified file name or a name without directory. In the latter case, the transformed file will be written into the source directory. If there is specified a target directory without a file name, all transformed source files will be written into the target directory using their original names.

-b BACKUP

A backup directory is necessary, if at least one source file will be overwritten by a transformation. If no file will be overwritten, no backups are made.

Examples:

tetra_cl -p exchange.ttp -s feuerbach.txt -b C:\Backup

"feuerbach.txt" will be overwritten by its transformed version. A backup of "feuerbach.txt" will be copied into "C:\Backup" before the transformation starts.

tetra_cl -p exchange.ttp -s feuerbach.txt -t bachfeuer.txt

The transformed version of "feuerbach.txt" will be written into "bachfeuer.txt" in the same actual directory.

tetra_cl -p exchange.ttp -s *.* -t ..\newdir

The transformed files will be written into the subdirectory "..\newdir". If this directory doesn't exist, it will be created automatically.

If there are several source files but only one target file, all results will be written into the single target file. This must be different from all source files.

-c CONFIGURE

Following on "-c", you can submit parameters as an identifier or string to the project, which are needed there before the start of any of the transformations. The parameter is read with the function ConfigParam there. A single config parameter is put for all files.

-x EXTRA

Following on "-x", you can submit parameters as an identifier or string to the project, which are needed there before the start of a certain transformation. The parameter is read with the function ExtraParam there. An extra parameter can be put per file.

-a ASK

If the optional parameter "-a" is written, you will be asked before each transformation:

transform: source to target ? (yes,no,all,cancel)

You have to answer by pressing the according first letter:

y	transform this file
n	don't transform this file
a	transform all files
c	cancel all transformations

-r RECURSIVE

By the optional parameter "-r" you can force a recursive search for source files in all subdirectories.

9.6 Keyboard shortcuts

Help

Calls this help	F1
Regex Test	F2

File operations

CTRL + O	Open file
CTRL + S	Save file

Navigate

SHIFT + F5	goto startrule

Compile

F5	Parse startrule

Executing and debugging

F6	Next token
F7	Step into
F8	Step over
F9	Start
F10	Execute
F11	Transformation manager
CTRL + F12	Reset
CTRL + SHIFT + Digit	Set/remove breakpoint
CTRL + Digit	Jump to breakpoint

Repository

CTRL + ALT + I	Parse/test single script
CTRL + ALT + P	Parse/test connected scripts
CTRL + ALT + T	Parse/test all scripts
CTRL + ALT + N	New script
CTRL + ALT + A	Accept changes
CTRL + ALT + C	Cancel changes
CTRL + ALT + D	Delete script
Delete	Delete selected script of the list
CTRL + ALT + M	Comment
SHIFT + F3	Find next occurrence of the selected text (in all scripts)

Clipboard

The following keyboard shortcuts will work only, if a script is selected in the list of all scripts. If however an editor is active, the shortcuts have their usual text-processing function.

STRG+Insert copies the actual script into the clipboard. From there it can be reinserted into the same project with a new name, or it can be inserted into a different project, which is opened in a second instance of the TextTransformer.
Shift+Insert inserts a script from the clipboard into a project. (You have to accept the insertion and if there is a name conflict, you have to rename it.)

9.6.1 Block commands

Inside of the different edit fields (not in the comment field) following keyboard shortcuts for block command can be applied:

Shortcut	Action or command

CTRL+K+B	marks the beginning of a block
CTRL+K+I	indents a block
CTRL+K+K	marks the end of a block
CTRL+K+L	selects the actual line as block
CTRL+K+N	converts the block to upper case
CTRL+K+O	converts the block to lower case
CTRL+K+T	selects a word as block
CTRL+K+U	'unindents' a block
CTRL+K+Y	deletes the selected block

Part X

10 Scripts

A TextTransformer project can consist of four kinds of scripts:

Token definitions that describe the lexical units of the input as regular expressions are located on the token page. (If they are literals, they can immediately be defined inside of the grammar rules.)

Grammar rules (productions) that describe the syntactical structure of the input. These rules are located on the production page.

C++ code that describes the translation of the input into a target language. This code is embedded into the grammar rules by use of special parenthesis.

Test scripts on the test page of the TextTransformer.

For simple projects the definition of grammar rules is sufficient.

The names of all production, element and token scripts of a project have to be different to each other.

10.1 Token definitions

On the token page you can define tokens.

The definition of a token is done inside of a form.
Literal tokens can be defined directly inside of a production too.

On the following pages the syntax of regular expressions will be explained.
At first a simple subset of regular expressions will be presented: the literal expressions. Then the regular expressions will be presented in detail.

10.1.1 Input mask for a token

The mask for a token definition has the following fields:

Name: unique name
Return type: C++-type
Parameter: C++-Parameter declaration
Comment: arbitrary comment
Text: Token script
Semantic action: Instructions

Name and text are needed. If one of these fields is empty, the script will not be accepted and you can't write a comment.

10.1.1.1 Name

Each token script must have a name. This name is used in production scripts to denote the token. A name can be constructed of the alphanumeric characters and the underscore, but the latter may not be at in first place of the name.

Examples: IDENTIFIER, int_value, UB40

Each new name must differ from all other names of tokens, productions, functions and variables by at least one character.

10.1.1.2 Return type

The commands in the field *semantic actions,* which are executed immediately after the recognition of the token, can return a value.
For the syntax the same applies, what is said for the return type of productions.

10.1.1.3 Parameter declaration

In the parameter field you can declare parameters for the token. The parameters can be used by the semantic actions.
For the syntax the same applies, what is said for the parameters of productions.

10.1.1.4 Comment

A comment to the token script can be shown in the yellowish field. Temporary this field is also used to show error messages.

To change the comment, use the button. A dialog will be opened, where you can write the new text.

10.1.1.5 Text

In the field *Text* the token definition as regular expression is formulated. In the simples form

the whole expression is written into the first line of the field.

This is possible in each case and in most cases this will be done so. But you can increase the readability of complex expressions by

writing them into several commented lines.

Hereby the parts of the expression must be at the beginnings of the different lines and the conjunction of the lines is made by a backslash " \" preceded of at least one white space. Each **conjunction character "\"** and also the last part of the regular expression may be followed by a line comment. A **line comment** begins with two slashes **"//"** and covers the rest of the line. The text must be separated from line comments by at least on white space or a line break. The following notations are equivalent:

```
\w+::\w+ // class and function name

\w+ \  // class name
::\w+  // function name

\w+ \
// class name
::\w+
// function name
```

10.1.1.6 Semantic action

The field *Semantic action* is used for c++ instructions that specify how the parser reacts to the recognition of the token. Even if a token is often used in the grammar, this reaction must be specified only once. For example, like a bilingual dictionary the translation of a token text can be given here:

```
Text:    "Hello world"
Action: out << "Hallo Welt";
```

For the syntax of the instruction the same is applied as described for semantic actions in productions.

If you don't use braces, the project options for double braces are applied.

For token actions no nodes will be created in the syntax tree. So the tree becomes clearer.

It isn't possible to define a transitional action for a token if an action is assigned to it as described here.

10.1.2 Literals

Each special word of a text, each number and generally each part of a text can be considered as an individual literal token. A literal simply is a special sequence of characters.
For example the word "TETRA" is a 'T" followed by an 'E' a 'T', 'R' and an 'A'.

According to their simplicity and their importance for syntactical analysis literals have not to be defined separately on the token page. You can define them directly inside of a production. This is possible in two ways:

1. by inclusion of the text in quotation marks, e.g. "TETRA"
2. by putting an underscore in front of the text, e.g. _TETRA,

In the second case a named literal token is produced that is inserted on the token page automatically. The special advantages of named literal tokens are discussed separately. The simple literal tokens usually suffice. For example a rule to parse a salutation could look like

```
( "Mr" | "Mrs" ) name
```

Hereby "Mr" and "Mrs" are meaning themselves, while *name* could denote a different regular expression or a production.

Inside such a token each character means itself. That holds not generally for regular expressions. For example the smiley

";-)"

defined in the syntax of regular expressions looks like:

";\-\)"

Here the hyphen and the parenthesis have a Meta meaning, so that they must be preceded by a backslash to get back their originally meaning.

Some characters, which could be represented otherwise, such as line breaks, can be used as escape sequences also within the definition of literal tokens.

10.1.2.1 Named literals

Sometimes it is advisable to define named literal tokens instead of using simple literal tokens. Either this can be done directly on the token page or you simply can define them within a production too, by putting an underscore in front of the literal text. In this case a new token is inserted on the token page automatically, as soon as the production is accepted. If e.g. the expression _TETRA occurs in the production, then the following token is produced:

Name	_TETRA
Definition	TETRA

Since special characters aren't permitted in script names, this way no tokens which include special characters can be defined. However, it is possible to change the definition of a named literal afterwards correspondingly. E.g.:

Name	_TETRA
Definition	**TETRA**

If the name of a token starts with an underscore, all characters occurring in the definition are interpreted literally. So the stars '*' don't have the meaning of repeat marks here.

The advantages of named literal tokens are that they

1. have a unique name,
2. can be associated with a semantic action easily and
3. these names also are availably in the produced c++ code.

to 1.
Misspellings which could occur if the same literal is used in different places of the grammar are avoided.

to 2.
If the same semantic action shall always be executed when finding a certain literal token in the text even if it occurres in different places of the grammar, then it is appropriate to make use of the possibility of connecting a token directly with an action. E.g. it often makes sense at the generation of parse trees to insert the texts of the literal tokens as leafs in the tree in always the same way.

to 3.
A character string is produced at the generation of c++ code from a project for every named literal. These strings can be used also in the code of the program, which uses the parser. For example:

```
const char _protected[] = "protected";
```

10.1.3 Regular expressions

The TextTransformer uses the regular expression library of Dr John Maddock. The syntax of the expressions is the same as described there - with little restrictions.

Following elements are used to define regular expressions

- Characters by code
- Special characters
- Sets of characters
- Character classes
- Locale dependant features
- Wildcard
- Anchors

Concatenation
Groupings
Alternatives
Repeats
Macros

10.1.3.1 Single characters

All characters match themselves, if the don't have a special Meta-meaning.. So e.g. the letter 'A' in the source text is recognized with the character 'A' in the regular expression and if a regular expression contains the string "hello", it will recognize the word "hello"

10.1.3.2 Meta-characters

In contrast to literals, where all characters designate themselves, regular expressions use some special Meta characters. Meta characters are used to define sets of characters, groupings and sequences.

Following characters are **Meta characters**:

'.', '|', '*', '?', '+', '(', ')', '{', '}', '[', ']', '^', '$' und '\'

If a Meta character in its literal meaning is needed, is must be preceded by the backslash '\'.

Quoting escape

The escape sequence \Q begins a "quoted sequence": all the subsequent characters are treated as literals, until either the end of the regular expression or \E is found. For example the expression: \Q*+\Ea+ would match either of:

```
\*+a
\*+aaa
```

10.1.3.3 Special characters

The visual representation of other nongraphic characters is possible by escape sequences.

The rules for the use of hexadecimal numbers are modified a little in contrast to the use in general character strings. Only two hexadecimal numbers are interpreted as a characters if they aren't parenthesized.

\xdd A hexadecimal escape sequence - matches the single character whose code point is 0xdd
\x{dddd} A hexadecimal escape sequence - matches the single character whose code point is 0xdddd.

\N{Name} Matches the single character which has the symbolic name *name*. For example \N{newline} matches the single character \n.

10.1.3.4 Sets of characters

A set of characters can match any single character that is a member of the set.
Sets are delimited by "[" and "]" and can contain literals, character ranges or predefined character classes.

The characters ".|*?+(){}$", which have a Meta meaning within the regular expressions, have their literal meaning within the definition of character classes, i.e. no backslash must be put in front of them here.

Set declarations that start with "^" contain the compliment of the elements that follow.

Examples:

Character literals:

"[abc]" will match either of 'a', 'b', or 'c'.

"[^abc]" will match any character other than 'a', 'b', or 'c'.

Character ranges:

"[a-z]" will match any character in the range 'a' to 'z'.

"[^A-Z]" will match any character other than those in the range 'A' to 'Z'.

Combinations:

All of the above and character sets and symbolic names can be combined in one character set declaration, for example:

[[:digit:]a-c[.NUL.]].

To include a literal '-' in a set declaration then: make it the first character after the opening '[' or '[^', the endpoint of a range precede with an escape character as in '[\-]'. To include a literal '[' or ']' or '^' in a set then make them the endpoint of a range, or preceded with an escape character.

10.1.3.5 Character classes

Character classes are denoted using the syntax "[:classname:]" within a set declaration, for example "[[:space:]]" is the set of all white space characters.
[[:digit:],] is the set of all digit and the comma.

The available character classes are:

alnum	Any alpha numeric character; *alpha* and *digit* (*)
alpha	Any alphabetical character a-z and A-Z, umlauts etc. (*)
blank	Any blank character, either a white space, a non-breaking space (decimal 160) or a tab
cntrl	Any control character
digit	Any digit 0-9
graph	Any graphical character; all other except *cntrl*
lower	Any lower case character a-z (*)
print	Any printable character, *graph* and *blank*
punct	Any punctuation character
space	Any white space character (space, tabulator, carriage return, line feed...)
upper	Any upper case character A-Z (*)
xdigit	Any hexadecimal digit character, 0-9, a-f and A-F
word	Any word character - all alphanumeric characters plus the underscore (*)

(*) according to the local settings on your computer other characters might be recognized too. Try it in the dialog for the calculation of character classes!

There are some shortcuts that can be used in place of the character classes

\w	[:word:]
\W	^[:word:]
\s	[:space:]
\S	^[:space:]
\d	[:digit:]
\D	^[:digit:]
\l	[:lower:]
\L	^[:lower:]
\u	[:upper:]
\U	^[:upper:]

10.1.3.6 Locale dependant features

There are some country or language specific definitions for regular expressions. This concerns their use in TextTransformer only marginally, because, at the moment only the German localization - if installed on your computer - or otherwise the English localization is supported and the TextTransformer IDE always uses the ANSI-character set. Nevertheless, in special cases these features may be useful and a knowledge can be also helpful for an extended use of the generated code.

Collating elements
Equivalence classes

Collating Element Names

10.1.3.6.1 Collating elements

An expression of the form [[.col.]] matches the collating element col. A collating element is any single character, or any sequence of characters that collates as a single unit. Collating elements may also be used as the end point of a range, for example: [[.ae.]-c] matches the character sequence "ae", plus any single character in the range "ae"-c, assuming that "ae" is treated as a single collating element in the current locale.
Collating elements may be used in place of escapes (which are not normally allowed inside character sets), for example [[.^.]abc] would match either one of the characters 'abc^'.
As an extension, a collating element may also be specified via its symbolic name, for example:

```
[[.NUL.]]
```

matches a NUL character.

10.1.3.6.2 Equivalence classes

An expression of the form[[=col=]], matches any character or collating element whose primary sort key is the same as that for collating element col, as with colating elements the name col may be a symbolic name. A primary sort key is one that ignores case, accentation, or locale-specific tailorings; so for example [[=a=]] matches any of the characters: a, à, á, â, ã, ä, å, A, À, Á, Â, Ã, Ä and Å. Unfortunately implementation of this is reliant on the platform's collation and localisation support; this feature can not be relied upon to work portably across all platforms, or even all locales on one platform.

10.1.3.6.3 Collating Element Names

Digraphs (Connection of two letters to a sound)

The following are treated as valid digraphs when used as a collating name:

"ae", "Ae", "AE", "ch", "Ch", "CH", "ll", "Ll", "LL", "ss", "Ss", "SS", "nj", "Nj", "NJ", "dz", "Dz", "DZ", "lj", "Lj", "LJ".

POSIX Symbolic Names

The following symbolic names are recognised as valid collating element names, in addition to any single character:

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 Wildcard

The dot character "." matches any single character. You can interpret this as a set containing all characters.

10.1.3.8 Anchors

Warning: Anchors only have an effect if the expression is target of a Skip node and the effect of anchors depends on the ignorable characters set in the project options.

An anchor is something that matches the null string at special text positions.

Line anchor:

'^' matches the null string at the start of a line, when used as the first character of an expression, or the first character of a sub-expression.

'$' matches the null string at the end of a line, when used as the last character of an expression, or the last character of a sub-expression.

Word anchor:

"\<" matches the null string at the start of a word.

"\>" matches the null string at the end of the word.

"\b" matches the null string at either the start or the end of a word.

"\B" matches a null string within a word.

Buffer anchor

"\`" matches the start of a buffer.

"\A" matches the start of the buffer.

"\'" matches the end of a buffer.

"\z" matches the end of a buffer.

"\Z" matches the end of a buffer, or possibly one or more new line characters followed by the end of the buffer.

A buffer is considered to consist of the whole sequence passed to the matching algorithms.

Remark:

In the TextTransformer "\A" precedes all regular expressions - when not used inside a SKIP definition - to assert, that the next match will begin at the current text position. So an anchor like "^" has no use. The search begins at the beginning of the actual buffer and the regular expression "does not know" where the buffer begins in the text.
But for SKIP nodes the use of an anchor "^" can make sense:
If a token is defined as:

LINE_START_WORD = ^\w+

and the grammar defines a SKIP symbol:

SKIP LINE_START_WORD

the first line of the following text will be skipped:

" Wort1 not at line begin
Wort2 at line begin"

"Wort2" will be recognized as LINE_START_WORD.

10.1.3.9 Concatenation

Hitherto only elements of regular expressions were explained, which matches single characters. From these and elements explained below, complex regular expressions can be constructed, to match lists of characters like words and numbers etc. The simplest manner for this is to write one element after the other. So each element recognizes one character of the whole list.

So the word "TETRA" consists of the regular elements "T", "E", "T", "R" and "A" But the same word would also be recognized by "[A-Z]E[^0-9]RA". This expression is more general than the first and also would match texts like "AE&RA" or "HEDRA".
The principle is the same: The x'th element of the complex regular expression matches the x'th character.

10.1.3.10 Groupings

Parentheses serve two purposes:

1. To group items together into a sub-expression

If expressions are grouped into a sub-expression, operators like the repetition operator, can be applied to the whole group. For example the expression "(ab)*" would match all of the string "abababab".

2. To mark what generated the match

When the whole expression has matched, both informations can be accessed (via xState): on what the whole expression matched and on what each sub-expression matched. Sub-expressions are indexed from left to right starting from 1; sub-expression 0 is the whole expression. The according parts of text are obtained by

```
str s = xState.str(index);
```

It is permissible for sub-expressions to match null strings. If a sub-expression takes no part in a match - for example if it is part of an alternative that is not taken - then xState.str(index) will return an empty string.

10.1.3.11 Alternatives

A regular expression can contain alternatives. That means, the whole expression matches the text, if one of the alternatives matches. Alternatives are separated by the pipe character "|".

Examples:

"a(b|c)" matches "ab" or "ac".

"abc|def" matches "abc" or "def", but not "abdef".

At the last example you can see, that each alternative contains the largest possible sub-expression (in contrast to repetitions see below). "abc|def" is not "ab" followed by "c" or "def", but "abc" or "def".

10.1.3.12 Repeats

Any atom (a single character, a marked sub-expression, or a character class) can be repeated with the *, +, ?, and {} operators.

*****

The * operator will match the preceding atom zero or more times, for example the expression a*b will match any of the following:

```
b
ab
aaaaaaaab
```

+

The + operator will match the preceding atom one or more times, for example the expression a+b will match any of the following:

```
ab
aaaaaaaab
```

But will not match:

```
b
```

?

The ? operator will match the preceding atom zero or one times, for example the expression ca?b will match any of the following:

```
cb
cab
```

But will not match:

```
caab
```

{}

An atom can also be repeated with a bounded repeat:

a{n} Matches 'a' repeated exactly n times.

a{n,} Matches 'a' repeated n or more times.

a{n, m} Matches 'a' repeated between n and m times inclusive.

For example:

```
^a{2,3}$
```

Will match either of:

```
aa
aaa
```

But neither of:

```
a
aaaa
```

It is an error to use a repeat operator, if the preceding construct can not be repeated, for example:

```
a(*)
```

Will raise an error, as there is nothing for the * operator to be applied to.

All repeat expressions refer to the shortest possible previous sub-expression: a single character; a character set, or a sub-expression grouped with "()" for example.
The expression "ba*" doesn't match "baba".

10.1.3.13 Macros

The names of already defined token can be used for the definition of other token. To do this, the name must be included into the braces '{' and '}'. Now this expression can be situated inside of a new token definition. When parsing the new definition, the TextTransformer will remove the braces and substitute the token name by its text.

Example:

```
SPACES = [ \t]*

DECLARATOR =
                    (((\w+::)*\w+)::)?(\w+) \ //Scope(s) and name
                    {SPACES} \              // optional spaces
                    \([^)]*\)               // Parameter
```

Internally the TextTransformer will collapse the line to one:

```
(\w+::)?(\w+)::(\w+)[ \t]*\([^)]*\)
```

10.1.3.14 boost regular expression library

The TextTransformer uses the regular expression library of Dr John Maddock at http://www.boost.org, which is a part of the whole boost library. The syntax of the expression is the same as described there as POSIX-Extended Regular Expression Syntax.
The boost regular expressions can be modified by flags. With two exceptions the default flags (= extended) are kept.

```
const tt_syntax_option_type _boost_regex_normal =
      boost::regex_constants::extended
       & ~boost::regex_constants::no_escape_in_lists
       & ~boost::regex_constants::collate;
```

The flag "no_escape_in_lists" is negated. So a backslash has to be put in front of itself inside of the definition of a character set.

The flag "collate" is negated, to be able to define character sets in the order, by which the characters are listed in the ANSI-table.

The flag "extended" specifies that the grammar recognized by the regular expression engine is the same as that used by POSIX extended regular expressions in IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Base Definitions and Headers, Section 9, Regular Expressions (FWD.1).

When the expression is compiled as a POSIX-compatible regex then the matching algorithms will match the first possible matching string, if more than one string starting at a given location can match then it matches the longest possible string.

This matching algorithm is essential for the TextTransformer. So unfortunately non-greedy repeats and look-ahead asserts aren't supported. Backreferences aren't disabled explicitely, but normally they will not work correctly, as the enumeration of subexpressions is moved in SKIP-expressions. Backreferences have to be avoided.

10.1.4 Predefined tokens

If you are in the list of tokens on the left side of the token page with your mouse, you will get a list of predefined tokens, by right mouse click. If you choose one of these tokens, a new script with filled name and text field will be opened. You now can modify it, or accept it as it is.

The predefined token are divided into the following categories:

- Identifier
- Words
- Numbers

Quotes
Dates
Comments
Ignorable
Line break
Binary null
Addresses
Data field (Character class calculator)

10.1.4.1 Identifier

The following predefined tokens can be inserted in a project by a pop-up menu.

ID: `[a-zA-Z_]\w*`

Identifiers beginning with a character of the alphabet or the underscore and followed by an arbitrary number of alphanumeric characters or the underscore (= \w). It is important, that an identifier cannot begin with a digit. Otherwise it would recognize numbers too. Mostly it is recommended, to define an extra token for numbers.

URI_WS_DELIM
URI_QUOTE_DELIM
URI_ANGLE_DELIM

This are regular expressions for Uniform Resource Identifiers (URI), as for example website addresses. URI's are described in RFC 3986

http://www.apps.ietf.org/rfc/rfc3986.html

The three expressions are variants of an expression given on the mentioned page. As opposed to the latter these expressions don't recognize any empty text. A URI has to start with the "scheme" expression described there, i.e. with characters on which a colon follows. In addition, a URI has to be delimited from the surrounded text. The three expressions given here are different in the way of this separation.

URI_WS_DELIM : (([^:/?#]+):)(//([^/?#\\s]*))?([^?#\\s]*)(\\?([^#\\s]*))?(#([\\s]*))?

This URI is delimited by white spaces from the rest of the text. So the URI may not contain white spaces. For example *URI_WS_DELIM* recognizes the following text:

```
http://www.ics.uci.edu/pub/ietf/uri/#Related
```

URI_QUOTE_DELIM : "(([^:/?#]+):)(//([^/?#"]*))?([^?#"]*)(\\?([^#"]*))?(#([^"]*))?"

This URI is delimited by double quotes from the rest of the text. So the URI could have been written in a manner, that it contains white spaces. For example *URI_QUOTE_DELIM* recognizes the following text:

```
"http://www.ics.uci.edu/pub/ietf/uri/#Related"
```

URI_ANGLE_DELIM : <(([^:/?#]+):)(//([^/?#>]*))?([^?#>]*)(\\?([^#>]*))?(#([^>]*))?>

This URI is delimited by angle brackets from the rest of the text. So the URI could have been written in a manner, that it contains white spaces. For example *URI_ANGLE_DELIM* recognizes the following text:

```
<http://www.ics.uci.edu/pub/ietf/uri/#Related>
```

Remark:

The following sections are recognized by the sub-expression in the example above:

$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

where <undefined> indicates that the component is not present. Therefore, we can determine the value of the five components described in RFC 3986 as

scheme = $2
authority = $4
path = $5
query = $7
fragment = $9

10.1.4.2 Words

The following predefined tokens can be inserted in a project by a pop-up menu.

The classes of possible recognitions of the words of different languages are overlapping. You only

should use one language in one project or be sure what you are doing.

WORD_EN: `[A-Za-z]+`

WORD_GE: `[A-Za-zäöüÄÖÜ][A-Za-zäöüÄÖÜß-]*`

WORD_FR: `[A-Za-záàâéèêíìîóòôúùûÀÁÂÈÉÊÌÍÎÒÓÔÙÚÛÇçŒœ]+`

WORD_SP: `[A-Za-záéíóúÁÉÍÓÚñ]+`

English, German, French or Spain word. The regex consist in one or more repeats of the letters on the according alphabet. The letters [a-z) and [A-Z] (= [[:alpha:]] are contained in all alphabets. (German words may not begin with 'ß' and may contain hyphens.)

NAME_INT: `[[:alpha:]][-[:alpha:]]*`

International name, which may contain hyphens.

10.1.4.3 Numbers

The following predefined tokens can be inserted in a project by a pop-up menu.

INT: `\d+`

A natural number (integer) simply consists in one or more digits. The class of all digits is predefined as '\d'.

REAL: `(\d+\.\d*|\.\d+)([eE][\+\-]*\d+)?`

A real number, either begins with a dot or with a dot preceded by an integer. The dot is followed by an arbitrary number of digits and optionally by an exponent expression. A dot without an adjoining digit is not a real number.

Examples: .1; 1.;1.1; 3.14, 6.626E-34

HEX_PAS: `\$[[:xdigit:]]{1,8}`

A hexadecimal number as used in the programming language pascal: a dollar character followed by up to eight digits or letters from 'A' to 'F' or 'a' to "f'.

HEX_CPP: `\$[[:xdigit:]]+`

A hexadecimal number as used in the programming language c++: "0x" followed by several digits or letters from 'A' to 'F' or 'a' to "f'.

10.1.4.4 Quotes

The following predefined tokens can be inserted in a project by a pop-up menu.

STRING: `"([^"\\\r\n]*(\\.[^"\\\r\n]*)*)"`

String, beginning and ending with a double quote (\"). The string may contain other double quotes, if they are preceded by backslashes.

"This expression was \"Friedl\" optimized"

The first sub-expression contains the characters inside of the outer double quotes, that means, by xState.str(1) you can access this text directly.

A string as defined here cannot extend the end of line.

Remark: The expression is optimized according to Friedl's scheme

CHAR_CPP: `'(\\?.)'`

Characters are included in quotes as usual in c++. Preceded by a backslash, they get a special meaning, as e.g. '\n': the line feed.

10.1.4.5 Dates

The following predefined tokens can be inserted in a project by a pop-up menu.

The classes of possible recognitions of the following dates are overlapping. You only should use one of them in one project or be sure what you are doing. The expressions are kept general. So are recognized next to 4-digit also 2-digit dates and different separators are allowed. For a concrete application the expressions should be specified as far as possible.

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
```

This expression recognizes date like: 03.02.56; 3.2.1856; not valid dates like 31.2.9999 are not excluded.

Remark: Each of the parenthesis of the expression contains exactly one part of the date. So day, month and year can be easily obtained by:

```
str sDay = xState.str(1);
str sMonth = xState.str(2);
str sYear = xState.str(3);
```

The regular expression is optimized according to the LL(1)-principle. It begins with alternatives, which exclude each other. For example the 31 days of a month must begin with a preceding 0 or by another digit. Each digit only can be followed by specific other digits.

alternative	begin	range
0 **[** 1-9 **]**	**0**	01, 02, 03, 04, 05, 06, 07, 08, 09
| **[** 12 **]** **[** 0-9 **]?**	**1**	1 oder 10 - 19
	2	2 or 20 - 29
| 3 **[** 01 **]?**	**3**	3 or 30 or 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
```

This expression is a simple rearrangement of the expression above. It recognizes dates like 56.03.02; 1856.3.2; not valid dates like 9999.31.2 are not excluded.
recognizes

10.1.4.6 Comments

The following predefined tokens can be inserted in a project by a pop-up menu.

They describe comments of different programming languages. Frequently they are use as parts of the ignorable characters. In this case they neither need nor can be used inside of the productions.

LC: `//[^\n]*(\n|\z)`

Line comment, beginning with \"//\" and extending till the end of the line or end of file.

BC_CPP: `/\*[^*]*\*+([^/*][^*]*\*+)*/`

Block comment (C++-style), beginning with "/*" and ending with "*/". Inside of block comments sequences of '*' are permitted, e.g..: /********/. It's difficult to understand the construction of this expression. It is derived in the book: J.E.F. Friedl:Regular expressions. ("[^/*]" here represents *special*.)

BC_PAS: `\(\*[^*]*\*+([^)*][^*]*\*+)*\)`

Block comment (Pascal-style), beginning with "(*" and ending with "*)". Inside of block comments sequences of '*' are permitted, e.g..: (********)

BC_HTML: `<!--([^-]|-+[^->]|->)*-+->`

HTML/XML block comment, beginning with "<!--" and ending with "-->".

10.1.4.7 Ignorable

The following predefined tokens can be inserted in a project by a pop-up menu.
Then they can be set in the project options for the ignorable characters.

IGNORE_CPP:

```
(\s \// spaces
|/)      \// begin of a comment
/[^\r\n]*$   \// line comment
|\*[^*]*\*+([^/*][^*]*\*+)*/ \// block comment
)   \
)+
```

Line comments, block comments and spaces are ignored in c++-code.

IGNORABLE_PAS:

```
( \
\s \// spaces
|\{[^}]*\} \// {...}-comment
|\(\*[^*]*\*+([^)*][^*]*\*+)*\) \// (*...*)-comment
)+
```

{...} comments, (*...*) comments and spaces are ignored in Pascal code

IGNORE_XML:

```
(\s \// spaces
|<!--([^-]|-+[^->]|->)*-+->)+ // block comment
```

XML ignores block comments and spaces.

10.1.4.8 Line break

EOL: `\r?\n`

This definition matches WINDOWS and UNIX line breaks, if '\r' and '\n' are not set to be ignored in the project options.
In the editor all lines are made by "\r\n", even if the source file only contains '\n' line breaks.

As a predefined token *EOL* can be inserted in a project by a pop-up menu.

10.1.4.9 Binary null

NULL: `\x00`

NULL is defined as the character with the value null. The use of this token only makes sense when binary files are parsed. It cannot occur in text files. It marks the end of the file EOF there.

As a predefined token *NULL* can be inserted in a project by a pop-up menu.

10.1.4.10 Addresses

The following predefined tokens can be inserted in a project by a pop-up menu.

EMAIL:

```
[\w\.-]+ \// local part
@ \
([\w-]+\.)+ \ // sub domains
[a-zA-Z]{2,4} // top level domain
```

A usual e-mail address. The complete syntax of an e-mail address is quite complex and cannot be reasonably described with a single regular expression.

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 address, all four numbers restricted to 0..255

The address expressions are from an article of Vasant Raj:

http://www.codeproject.com/useritems/Regular_Expressions.asp

The expression for the e-mail address was simplified.

10.1.4.11 Data field

If the entry if in the pop-up menu of the predefined tokens: Data field is activated, the character class calculator appears. By this dialog expressions can be generated, which e.g. can recognize data fields of a database table.

Fields of a database table are frequently defined by a certain width: a number of characters. The characters to be used within the field are restricted too. E.g. a field can be defined for ten alphanumeric characters:

```
[[:alnum:]]{10,10}
```

10.1.5 Placeholder

Placeholder tokens get their meaning from the input text. Parts of text - mostly words -, which are recognized by a general regular expression, can be assigned to a placeholder token. If the same part of texts exists at a following position in the text again, the placeholder token now can recognize it.

You can assign an arbitrary number of literal expressions to a placeholder token. If several literals are assigned, they build a sequence of alternatives. Only one of them can match a text at a certain position.

Dynamic tokens are only tested, if they are expected as a possible alternative.

Independently of the project settings placeholder tokens always are case sensitive and the word boundary option is set.

A placeholder token is defined by the expression:

{DYNAMIC}

The expression *{DYNAMIC}* is a keyword and will **not** be interpreted as a macro by the TextTransformer.

The assignment of a literal expression is done in a semantic action by the command: *AddToken*.

Example:

Variables, which are declared in a programming code with a certain type, can be assigned to a placeholder for that type. In the code that follows, the type of the variable is known.

```
ID ::=  \w+
INTEGER ::= {DYNAMIC}

"int"
(
   ID
   {{
   AddToken( xState.str(), "INTEGER" );
   // assigns the found identifier to INTEGER
   }}
   ","
)*

";"
```

INTEGER

If *ID* recognizes the names : *i1, i2* and *i3*, at a following position *i1* or *i2* or *i3* can be recognized as an *INTEGER*. *INTEGER* now has the meaning of a token, which would be defined as: i1|i2|i3

Remark: The extension of a placeholder token by an additional literal has **no effect on** the recognitions of a preceding **SKIP** symbol.

10.2 Productions

To transform an input text, it must be analyzed according to its syntax. This analysis is done by means of rules ("productions"), which describe the syntax. In this productions instructions are embedded, which determine, how to construct a new text from the analyzed source.

A production may be considered as a function - more exact: as a specification for creating a function - that parses a part of the input text. By creating code, this specification will result in a real function. The routine can return a value and will constitute its own scope for parameters and other local components like variables and constants. These again, can be passed to other productions, which are called like functions inside the body of the first production. The called productions parse sub sections of the part of text, which is parsed by the calling production.

The definition of a production is done inside of a form.

10.2.1 Input mask for a production

The input mask for a production has the following fields:

Name:	unique name
Return type:	C++ variable type
Parameter:	C++ parameter declaration
Comment:	arbitrary comment
Text:	script text of the production

Name and text are needed. If one of these fields is empty, the script will not be accepted and you can't write a comment.

10.2.1.1 Name

Each production script must have a name. This name is used in other production scripts to denote the production.
A name can be constructed of the alphanumeric characters and the underscore, but the latter may not be at in first place of the name.

Examples: Const_declaration, line

Each new name must differ from all other names of tokens, productions, functions and variables by

at least one character.

10.2.1.2 Return type

In the field *return type* a type identifier must be specified only for the case, that instructions of the semantic actions of the production return a value. Otherwise a procedure of the type *void* is created automatically.

By according bracketing it is possible to determine, whether a return type shall be used inside the interpreter or shall be written to the produced source code or both.
For example, a *string* is returned inside the interpreter and a pointer is the return type of the produced source code, if you write the following:

```
{-str-} {_CToken*_}
```

Without brackets the project option for double braces "{{...}}" will be applied.
It is important, that the parts of interpretable and exportable code keep coherently.

Only the interpretable code is type checked by the TextTransformer. Other code will simply be copied into the generated code and the c++ compiler then might find errors.

If a return type is defined in code only for the export a default value must be given. This can be done here by appending a slash and the value. E.g.::

```
{_ CProduktion* _}/NULL
```

10.2.1.3 Parameter declaration

In the parameter field you can declare parameters for the production. The parameters can be used by the semantic actions. For example:

```
str& s, int i
```

For such a declaration the project option for double braces "{{...}}" is applied.
By according bracketing it is possible to determine, whether a parameter shall be used inside the interpreter or shall be written to the produced source code or both.
For example, a *string* could be declared for use inside the interpreter and a pointer for the produced source code, if you write the following:

```
{-str& s-} {_int i}
```

It is important, that the parts of interpretable and exportable code result in coherent parameter lists.

```
{-str& s-} {_int i_} {-double d-}
```

would result in a wrong list, because a comma is missing: str& s double d. Correct versions are:

```
{-str& s-} {_int i_} {-, double d-}
{-str& s,-} {_int i_} {-double d-}
{-str& s, double d-} {_int i_}
```

```
{_int i_} {-str& s, double d-}
```

Only the interpretable code is type checked by the TextTransformer. Other code will simply be copied into the generated code and the c++ compiler then might find errors.

10.2.1.4 Comment

A comment to the production can be shown in the yellowish field. Temporary this field is also used to show error messages.

To change the comment, use the button. A dialog will be opened, where you can write the new text.

10.2.1.5 Text

The text of a production defines a grammar rule. The syntax for this definition is leaned on the syntax of regular expressions. Grammar rules use some of the same Meta characters and introduce some new key words.

The following list cites all uses Meta characters:

| Separation of alternatives

(...) Grouping of expressions
?Option
* An optional repetition
+ A repetition

[…] Calling parameter of a production

{-...-} Semantic action inside the interpreter
{_..._} Semantic action to export into the generated code
{=...=} Exportable semantic action, which also is executed by the interpreter
{{...}} Semantic action for export or of the interpreter according to the option set

// beginning of a line comment

The additional key words are:

BREAK to quit a recognition loop
EXIT to finish parsing
EOF end of input (end of file)
SKIP jump to the next token, which may follow one of the alternates of SKIP
IF...ELSE...END Conditional execution

WHILE...END Conditional execution

10.2.2 Elements

Similar to tokens, which are connecting characters, productions connect tokens. Regular sequences or alternatives of token sequences can be formulated as a rule, that means as a production.

Definitions of productions are based on three kinds of elements:

1a) Literal tokens directly defined inside of a production
1b) Named tokens, defined on the token page
2. Other productions

Remark: The text, which a token matched, is accessible by xState.str()

About 1a) **Literal tokens directly defined inside of a production**

Simple words of the natural language or key words of a formalized language, punctuation marks, operators etc. can, but must not be defined on the token page. These literal token can be defined directly inside of a production by including them into quotes.

Examples:

```
"TETRA"      matches: TETRA
";"          matches: ;
```

If the quotation mark shall be used as a component of a literal token, then a backslash '\' must be put in front of character to distinguish it of the enclosing quotation marks. The backslash must as well be put in front a backslash.

```
"\""         matches: "
"\\"       matches: \
```

About 1b) **Named tokens, defined on the token page**

The definition of complex token was described above. Inside of a production the names of already defined tokens can be used.

About 2. **Other productions**

The definition of a production can be based on the definitions of other productions by using the names of them.

Example:

```
Production1 ::= "hello" | "good bye"
Production2 ::= "world"
Production3 ::= Production1 Production2
```

10.2.3 Concatenation

If one symbol directly is followed by a second symbol, they are chained. Further symbols can extend this concatenation.

```
"TETRA" "makes" "fun"
```

is the concatenation of the three token: "TETRA", "makes" and "fun".

Just as

```
"TETRA" "makes" emotion
```

emotion might be a token defined on the token page or a different production.
The TextTransformer will search for **emotion** as well in the list of all token as in the list of all productions. This is the cause, why all names as well of tokens as of productions must be different.

Semantic actions take part in the concatenation.

The concatenation has the highest preference of all operators of a production.

10.2.4 Alternatives

Alternatives are separated by the pipe character "|".

Examples:

"all" | "nothing" denotes an alternative occurrence of "all" and "nothing" in the input.

text | **number** denotes the alternative productions **text** and **number**

Special cases are **empty alternatives**. Empty alternatives make sense, to execute an action, if no other alternative matches. For example an error message might be written in this case:

```
"all" | "nothing" | {{out << "Error"; }}
```

Remark:
The first set of an empty alternative contains the special token **EPS** with the number 2. In the special case, that no further token follows the empty alternative in the grammar, inside of the token box of the debugger the expected token will be denoted by "EPS":

It is important, not to create an **inadvertent empty alternative**. This is the case in the following production:

```
X ::= (
       | "empty"
       | "size"
       | "clear"
      )
```

Each of the alternative token is preceded by a pipe character '|'. But in front of the first there is nothing, to which an alternative is formed. Equivalent one could have written:

```
X ::= (
         "empty"
       | "size"
       | "clear"
       |
      )
```

To prevent the inadvertent construction of empty alternatives, these formulations are disallowed. If you really want to create an empty alternative without calling an action, you either can make the whole set of alternatives optional:

```
( "all" | "nothing" )?
```

or you can construct an empty alternative with an empty action:

```
"all" | "nothing" | {{ /*Empty*/ }}
```

Remark: Empty alternatives are nullable.

10.2.5 Grouping

By means of the parenthesis '(' and ')' expressions can be grouped.
So the preference of concatenation can be eliminated.

Example:

("a" | "b") ("c" | "d")	matches	"a c", "a d", "b c" and "b d"
"a" | "b" "c" | "d"	matches	"a", "b c" and "d"

If you forget the preference of concatenation, this can lead to astonishing results, especially, if semantic actions are involved.

Example:

```
{{str s;}}
"a"
{{s = "a"; }}
| "b"
{{s = "b"; }}
```

will result in an error message for the last line: Unknown identifier: s
Implicitly the first three lines are a closed chain. Only inside of this scope the declaration of string s is valid. Correctly you have to write:

```
{{str s;}}
(
"a"
{{s = "a"; }}
| "b"
{{s = "b"; }}
)
```

A different example for this is the *Inner-production in the introduction*. There parenthesis around "b" | "c" are necessary.

10.2.6 Repeats

The syntax for repeats is analogous to the according syntax for regular expressions, but the operators operate on whole tokens, productions or groups of them.

+

A token, production or group followed by the plus character '+' matches any number of occurrences of the group in the text, but at least one.
The +-operator may not be applied on nullable structures.

*****

A token, production or group followed by the star character '*' matches any number of occurrences of the group in the text, including null occurrences.

?

A token, production or group followed by the question mark '?' matches an optional occurrence of the group in the text, that means null or one occurrence.

{}

You can specify the minimum and maximum number of repeats explicitly. Thus *A{2}* is the token or production A repeated exactly twice, *A{2,4}* represents *A* repeated between 2 and 4 times, and *A{2,}* represents *A* repeated at least twice with no upper limit. In contras to the according syntax of the regular expressions white spaces are allowed inside the *{}*.

Attention: if the minimum number of pattern isn't found in the text, there is a fault. If the pattern

occurs however, the in the text more frequently than specified by the maximum number, there is a fault only when the next text pattern isn't recognized differently.

Remark:
The WHILE structure offers the possibility of determining the number of repeats dynamically

10.2.7 BREAK

By means of the **BREAK** symbol loops - (...)* or (...)+ - can be left. If the parser finds the **BREAK** symbol inside of a loop, the loop is left and the parsing will continue with the symbols following the loop.

Example:

```
( "a" "b" "c"| "d" | BREAK)+  "e"
```

matches following texts:

"a b c d a b c e"
"d d d e"
"e"

```
(A | BREAK)+  is equivalent to (A)*
```

Because at the **BREAK** symbol the loop is left immediately, other nodes cannot follow the **BREAK** symbol. To connect an action with the **BREAK** symbol, the action must be written in front of the **BREAK**:

```
( "a" "b" "c"| "d" | {{out << "break";}} BREAK)+  "e"
```

The **BREAK** symbol must be written into the same production, where the loop is defined, which will be left by the **BREAK** symbol. Outside of a loop the **BREAK** symbol has no meaning. It is not possible to split the example above into two productions:

```
xxx ::=
( "a" "b" "c"| "d" | Break)+  "e"

Break ::=
{{out << "break";}} BREAK    // wrong
```

In these aspects the use of the **BREAK** symbol is similarly to the use of "break" in c++. Indeed, in the generated code, **BREAK** will be substituted by "break".

By means of the **BREAK** symbol you can analyze structures, which would be an irresolvable problem for a normal top down EBNF syntax based analysis. For example a text could be structured like

```
(";" "a" )+ ";" "b"
```

This text is syntactically valid, but causes the warning message:

";" is the start and successor of nullable structures

In this case the warning may not ignored. Parsing of the input:

"; a ; b"

leads to an error. After recognition of "; a" there is a conflict between a continuation by a new loop or by ";" "b". The TextTransformer in such cases chooses the first alternative. "a" will be expected and not "b".
By means of the **BREAK** symbol the production can be reformulated to

```
(";" ( "a" | BREAK ) )+ "b"
```

Now the input is recognized correctly. After recognition of the second semicolon at the beginning of the second loop the **BREAK** alternative will be chosen, the loop will be left and the following "b" will be recognized.
You may think, that a different reformulation of the first production would have had the same result:

```
(";" ( "a" )? )+ "b"
```

But this rule also would recognize texts, which were not intended originally. For example:

";;;b"

10.2.8 EXIT

If the parser meats the key word **EXIT**, the analysis of the text will be interrupted. The break is done in the same manner, as if the interpreter had executed a throw instruction.

EXIT can be followed by the additional key word **OK**. This signalizes a regular interruption. Without **OK** an error is signalized.

Because at the **EXIT** or **EXIT OK** the program stops immediately, other nodes cannot follow the **EXIT** or **EXIT OK** symbol. To connect an action with the **EXIT** symbol, the action must be written in front of the **EXIT**:

Example:

```
{{ out << xState.FileName() << " finished" << endl; }}
EXIT
OK
```

EXIT can be used in look-ahead parsers to finish the look-ahead. In this case the look-ahead parser is finished regularly without throwing an exception.

In the code, the TextTransformer generates, **EXIT** is realized by throwing an exception, if UseExcept

is not set to false and if not a sub parser or look-ahead parser is just executed.

```
throw tetra::CTT_Exit("EXIT, true);
```

This exception is **not** catched automatically in the generated code.

10.2.9 EOF

EOF denotes the "end of file" or "end of input". This special token is normally created automatically. It is contained in the follow set of the start rule and in the follow set of the nullable structures at the end of the start rule. If the start rule itself is nullable, **EOF** is contained also in its first set.

It is possible to use **EOF** explicitly inside of a production. Behind **EOF** no other tokens can follow. Up to the end of the program **EOF** may be followed only by nullable structures or the program should be interrupted by **EXIT**.

Example:

```
"a"
("b" | EOF EXIT)
"c"
```

Remark: The production parser of the TextTransformer uses **EOF**, to realize the reduced output and assignment instructions (e.g.: {{out << }}) for the return values of productions.

10.2.10 ANY

The symbol **ANY** denotes a single token from the set of all tokens used in the project which isn't used in the same production as an alternative to this ANY symbol. It can conveniently be used to parse structures that contain
arbitrary text. For example, the content of an exportable action can be skipped by:

```
"{_" ANY* "_}"
```

In this example the closing "_}" is an implicit alternative of the repeated *ANY* symbol. This means that *ANY* matches any terminal except "_}".

ANY recognizes only tokens which at least occur once explicitely within the rules dependent on the start rule: This point applies to every parse-system separately.
The following text will be recognized by the example above, if the literal tokens "int", "=" and ";" and an ID token for identifiers and a NUMBER token for numbers occur in the rule dependences

```
{_ int i = 3; _}
```

then is recognized as

```
"{_" ANY ANY ANY ANY ANY "_}"
```

whereby ANY represents the corresponding tokens respectively

```
"{_" "int" ID "=" NUMBER ";" "_}"
```

It is no problem, if "int" doesn't occur in the rules, because "int" could be recognized as *ID* too. But if *NUMBER* would not be used in the rule (depending from the start rule), it had to be introduced here as an explicit alternative:

```
"{_" (ANY | NUMBER )* "_}"
```

Otherwise the text would not be parsed.

10.2.10.1 Options

This section is only for specialists. It is recommended to the normal user to leave the default setting "no failure alternative for ANY" in the project options and to skip this section.

In the project options you can choose, whether the context of the production which uses an *ANY* symbol is taken into account at the calculation of its token set or not.

1. no failure alternative

In accordance with this option the alternatives of a production which uses the ANY symbol, is taken into account at the calculation of the token set of this symbol. Example:

```
Produktion1  ::= ANY | "a"
Produktion2  ::=  "b"
Produktion3  ::=  Produktion1  |  Produktion2
```

Because Produktion2 is an alternative to Produktion1 "b" is an alternative of *ANY*. *ANY* therefore recognizes all tokens apart from "a" and "b".

This option is the default for new projects since it might correspond to the intuitions of the user.

There is a problem in a special case, though. If the *ANY* symbol is in an nullable structure at the end of a production and this has different successors in different contexts, an unexpected behavior can result.

```
Any  ::=  ANY+
Production  ::=  "a"  Any  "b"  |  "c"  Any  "d"
```

In this case the text "a d b" isn't parsed. The first alternative would let expect this but because of the second "d" is excluded from the set of the tokens recognized by *ANY*
In such cases the TextTransformer generates a warning.

2. failure alternative

The calculation of the tokens recognized by *ANY* is more simple and therefore is a little faster if the context isn't taken into account. However, this has the serious disadvantage that e.g. *Produktion3* above doesn't compile. *ANY* then recognizes all tokens apart from "a" and therefore is in conflict with *Produktion2*.

This option can make sense to guarantee compatibility with ***Coco/R*** projects since *ANY* is calculated this way there.

10.2.11 SKIP

By means of the **SKIP** symbol sections of the text, for which there are no explicit rules can be skipped.

Remark: The skipped text is accessible by xState.str() or trim_right_copy(xState.str()).

The key word *SKIP* is a complex symbol. The meaning depends on the context. It depends on:

1. the alternatives

The *SKIP* alternative is chosen, if none of the alternatives matches the actual text. The *SKIP* alternative is chosenIn the production:

```
(
    "}"
  | "]"
  | SKIP
)*
```

if there is neither a "}" nor a "]" at the actual position of the input.

2. the possible followers

The *SKIP* symbol matches the text beginning at the actual position and ending at the position, where the text is matched by a symbol, which can follow the *SKIP* symbol. If there are competing followers, the match at the next position will be chosen.

For example

```
SKIP
(
    "]"
  | "}"
)
```

recognizes in the input

```
param1, param2 ] }
```

the text "param1, param2 ", because "]" follows immediately, while "}" follows at a later position. The rule above (point 1) will have the same result. In the first pass of the loop *SKIP* matches and in the second loop "]" matches. In the third loop "}" will be recognized too.
But it must be taken into account, that the match of *SKIP* depends on the follower of the loop itself. In the following context:

```
Startrule ::= Rule1 Rule2

Rule1 ::=
(
    "]"
  | "}"
  | SKIP
)*

Rule2 ::= "param2"
```

SKIP only would recognize "param1, ".

Supplementary explanations

Possible conflicts are treated differently depending on the options.

a) Isolation of SKIP and ANY

Occurrences of *SKIP* symbols must be isolated from each other. A *SKIP* may not have a second *SKIP* as alternative and a second *SKIP* may not follow it directly. The following isn't allowed:

```
(SKIP | ...) | (SKIP | ...)  // wrong
```

or

```
(SKIP | ...) ( SKIP | ...)  // wrong
```

ANY mustn't as well immediately follow on SKIP

```
(SKIP | ...) ( ANY | ...)  // wrong
```

b) SKIP-Repeats

The following cases aren't different, they have the same result:

SKIP?
SKIP*

A follower must be found at the actual position or at a later position. In the second case, *SKIP* is executed once. The case

SKIP+

differs in the circumstance, that there may not be a follower at the actual position, but there must be one at a later position. Again, the SKIP node is executed only once.

c) Konsumation of the ignored characters

The function of the SKIP symbol **doesn't depend on the project options**! E.g. SKIP EOL will recognize the line end, even if '\r' and '\n' belong to the character, which are to be ignored. While *xState*.str() after recognition of a normal token provides a text section, that stops before the following ignorable characters, this isn't the case at a text section recognized by *SKIP*. If the ignored characters are blanks, you can get the corresponding result by

```
trim_right_copy( xState.str())
```

d) no dynamic SKIP

The set of tokens, which can follow on *SKIP* is not dynamically changeable. If a placeholder token, which follows on *SKIP*, is augmented by a literal, this extension has no effect on the recognitions of the *SKIP* symbol.

e) **naming of skip nodes**

The name of a skip node is constructed of "SKIP" and the number, which represents the node in a first set. For example: SKIP12 is a skip node, which is registered in the first set of its superior node by the number 12.

10.2.11.1 Options

There are three options for the *SKIP* symbol which control the treatment of possible conflicts. Two of this can get predefined in the project options. If a not predefined option shall be used, then this can be expressed by appending a parameter to "SKIP". E.g.

```
SKIP[ F ]
```

1. no failure alternative

If the use of global scanners is set in the project options, it can happen that at the current position of the source text a token which isn't expected in the grammar is recognized. For example regular expression for identifiers is defined in many projects. If now

```
SKIP  "Welt"
```

is used to parse

```
Hello  world
```

"hello" would be interpreted as an identifier. According to the scanner algorithm, SKIP wouldn't be tested. Instead an error message would be produced. If no failure alternatives are permitted, however, "hello" becomes skipped and therefore the complete text would be parsed correctly.

Within a production this option can be set explicitly with the parameter NFA (no failure alternative):

```
SKIP[ NFA ]
```

There is a **problem** with **hidden alternatives** at this option, though. Example:

```
( ID ";"* )* SKIP
```

If a text starts with a semicolon, then it is skipped with SKIP. If the text starts with an identifier on which a semicolon follows, then the identifier is recognized correctly. However, the semicolon is skipped again. Although it is recognized at first it is not judged to be an alternative to SKIP. In this case the strict option 3 would be adequate for SKIP (see below)

2. allow skip destinations at the current position // experimentally

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.
A similar conflict as just described arises if a token which is the destination of *SKIP,* is already recognized at the current position of the source text.

```
SKIP ID
```

The word "Hello" in the text:

```
Hello world
```

then would be recognized as identifier. The option described above, not to allow failure alternatives, would not help, since the skip destinations aren't part of these alternatives. If they were added, an expression like

```
SKIP? ID
```

would become senseless. *SKIP* would always match on the text (if not at the end with no following identifier). So it is to recommend to amend the grammar correspondingly: With

```
SKIP? ID+
```

"Hello world" is parsed correctly. Experimentally there is the NF-option (no failure)

```
SKIP[ NF ]  // experimentally
```

This option possibly will be no longer available in future TETRA versions, if not desired by users.

3. strict generation of errors

Errors were always caused up to TextTransformer 1.5.0 in the cases represented above. This option forces the programmer to consider and to treat conflicts explicitly. The possibility of making SKIP optionally was already mentioned. One other possibility would be:

```
( SKIP | ID ) ID
```

A strict generation of errors can be forced in a production with the F parameter:

```
SKIP[ F ]
```

Note:

It shall be pointed out again that the mentioned conflict possibilities result from the option to use global scanners. A conflict with literals only can arise if also the option to test all literals is set and a literal is an immediate alternative to SKIP. Otherwise literals aren't tested at all.

10.2.12 IF...ELSE...END

Alternatives, which would produce a LL (1) conflict in another place, are allowed in an IF...ELSE...END structure. The progress of parsing isn't only determined by the next token here but is controlled by predicates too; e.g. a look-ahead in the current text (see remark below).

This structure has more exactly the form:

IF(boolean expression)
 if-branch
ELSE
 else-branch
END

The IF-branch and the ELSE-branch are arbitrary concatenations or groupings of tokens and semantic actions. The ELSE branch is optional. So a simple IF expression is possible too:

IF(boolean expression)
 if-branch
END

The boolsche expression only is evaluated, if the expected tokens is in the first set of the IF-branch. If the IF-branch cannot start with the next token, the ELSE-branch is executed, independently of whether the IF-condition is correct or not. If there is noe ELSE-branch, the structure is nullable.

The boolean expression always is interpretable and exportable.

Examples:

The simple structure can be used e.g. to resolve the conflict in the following rules:

```
Declaration     ::= Type ( IdentEqual )? QualIdent ";"
IdentEqual      ::= Ident "="
QualIdent       ::= Ident ( "." Ident )*

// e.g: "int i = xState.itg;" oder "int i;"
```

Here is a LL(1) conflict, as both *`IdentEqual`* and *`QualIdent`* are beginning with *`Ident`*. It can be resolved either by factoring out of *`Ident`*:

```
Declaration ::=
Type Ident
(
  ( "." Ident )*
| "=" QualIdent
```

```
)
";"
```

or you can write a look-ahead production, using *IdentEqual*:

```
Declaration ::=
Type
IF ( IdentEqual() )
     IdentEqual
END
QualIdent ";"
```

There are LL(1) conflicts too, which cannot be resolved so easy or not at all. Then the IF...ELSE...END structure has to be used.

As boolean expression you also can use a class variable:

```
IF ( m_bProfile )
(
  {{ double start = clock_sec(); }}
  Production
  {{ out << clock_sec() - start << " s" << endl; }}
)
ELSE
  Production
END
```

1. Remark:

The following structure is an **infinite loop**, if the condition is wrong:

```
(
  IF(Condition)
     Production
  END
)*
```

The expected token, which belongs to the first set of *Production* also belongs to the first set of the loop. Because this token cannot be consumed, the loop is executed again and again.

Instead you should write:

```
WHILE(Condition)
  ( Production )*
```

or

```
(
  IF(Condition)
     Production
  ELSE
    BREAK
  END
```

```
)*
```

In this respect the IF-construct of the TextTransformers isn't comparable with the IF-construct Coco/R, where the condition is set before the loop.

2. Remark:

If one of the branches is nullable, the **complete structure is regarded as nullable**. This can have an **unexpected consequence**. No matter what the condition yields in the following structure the token 'd' is always recognized, if it is next in the text. Also, if the condition isn't satisfied, no error results, if in this text doesn't follow 'c' but 'd'.

```
IF ( Condition )
  "a"?
ELSE
  "c"
END
"d"
```

10.2.13 WHILE...END

Similar to the IF structure in a WHILE...END structure the decision whether a branch is executed is done by a boolean expression.

This structure has more exactly the form:

WHILE(boolean expression)
 while branch
END

The while-branch is a concatenation or grouping of tokens and semantic actions.

The boolean expression always is interpretable and exportable.

Example

```
NUMBER
{{
int i = 0, iCount = xState.itg();
}}
WHILE(i < iCount)
ReadData  {{ i++; }}
END
```

At first the number of following data records is read and then by *ReadData* the data records are read

themselves.

Example:

```
EmptyBracket ::= "[" "]"
NonEmptyBracket ::= "[" IDENT "]"

WHILE ( EmptyBracket() )
   EmptyBracket {{ iEmptyBracketsCount++; }}
END
NonEmptyBracket
```

The production *EmptyBracket* is used as well for parsing as for looking ahead. The empty brackets are counted until the first not empty bracket appears.

10.2.14 Actions

Before and after the recognition of a token or the call of a production semantic actions can be executed.(In very special cases this is possible during the recognition too.) Mostly these actions are used to process the last recognized section of text. The instructions, which shall be executed, are inserted into the text of a production. To distinguish them, they are included in special brackets. The instructions are formulated in c++.
The TextTransformer can interpret a sub set of c++ directly. But when generating source code c++ can be used without limitations. The code is just copied in this case. To distinguish between code, that shall be checked and executed internally and code, that shall be exported, there are different kinds of brackets.

{-...-} Semantic action inside the interpreter
{_..._}Semantic action to export into the generated code
{=...=} Exportable semantic action, which also is executed by the interpreter
{{...}} Semantic action for export or of the interpreter according to the option set

The sub set of c++, which the TextTransformer can interpret is presented in the next chapter

A typical example of an action is to copy the last recognized section of text into the output

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

The syntax of the semantic actions is checked by the TextTransformer only in so far, as they are intended for the interpreter. Only the c++ compiler will check code that simply will be copied into the generated parser.

The different kinds of brackets cannot be applied in the conditions of IF-structures and WHILE-structures. This code always is both: interpretable and exportable.

The project EditProds demonstrates the parallel use of semantic code for the interpreter and different code for the export.

10.2.14.1 Transitional action

Besides the semantic actions there also is a type of very special "syntactic" actions. A transitional action is executed after a token was accepted and before the next is recognized. This is the ideal time to insert of new dynamic tokens with *AddToken*, since the new token is then already available before the determination of the next token Transitional actions are necessary too, to assure that a production behaves just the same at its use for look-ahead in the text as normally. A divergent behavior could arise if during parsing new dynamic tokens are produced or if the text scope is changed. But if such a change is executed as a transition action, it is executed also during the look-ahead temporarily. Nevertheless, the set of the dynamic tokens and the text scope will remain the same before and after the look-ahead.

As transitional actions primarily are considered the functions: *AddToken*, *PushScope* and *PopScope*. As transitional actions they are executed, if they are appended to the expression for a token in conjunction with a dot.

Examples:

```
ID.AddToken(xState.str(),   "CLASS")
ID[n].AddToken(xState.str(),   "CLASS")
ID[n].AddToken(xState.str(),   "CLASS",   "NewScope").PushScope("NewScope");
```

It isn't possible to define a transitional action for a token, if a "normal" action is also assigned to it.

At the generation of C++ code the action only is copied, i.e. e.g. no automatic adaptation to the character type is carried out.

An interesting application is the recognition of constructors of c++ classes. Such constructors are indicated by an identifier followed by two colons and the same identifier again, e.g. "CParser::CParser". Normally the repeated identifier can easily be recognized by semantic code. However, such code isn't executed at a look-ahead. In this case you could defines a placeholder token "CLASS" and add the identifier to it during it's first recognition. The second occurrence is then recognized by the CLASS token.

```
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 Calling parameters

If a production calls another production the required parameters must be passed. Variables are declared inside of a semantic action. The name of such a variable then can be included into braces "[...]", which are following the name of the called production.

Example:

The production *Comment* may have the parameter str& xsComment.

Name: Comment
Parameter: str& xsComment
Text: …

A second production *Script* could call the *Comment* production. So the str parameter must be declared, before *Comment* can be called:

Name: Script
Parameter: …
Text: {{ str s;}} Comment[s] ...

10.3 Class elements and c++ instructions

While parsing the input simultaneous the recognized sections of text can be processed. Before and after each step of recognition semantic actions are executed. A semantic action consists of c++ instructions.
In contrast to programming languages, which have to translate instruction to binary code understandable for the computer and then to build a new executable from these binaries, in the TextTransformer a lot of instructions can directly be executed, that means: they are **interpretable**.

The integrated interpreter of the TextTransformer masters

1. A sub set of the instructions of the programming language c++
2. Several functions, which are created specially for the use in TETRA

3. Formatting instructions

4. Several instructions, to access the actual state of the parser

5. User defined functions and variables (class elements)

10.3.1 Input mask for class elements

Definitions of new functions and of class variables by the user are based on the instructions listed in 1 - 4. Inside of the parser class, which the TextTransformer generates, these functions and variables are class elements.
The functions and variables and function-tables are indicated in the list of class elements by different symbols:

 function
 variable
 function-table

For each of these class elements there are the usual input fields:

Name:	unique name
Type:	C++ variable type
Parameter:	C++ parameter declaration (for functions only)
Comment:	arbitrary comment
Text/Initialization:	script text of a function or initialization of a variable

For functions name and text are needed, for a variable a name suffices. If one of the needed fields is empty, the script will not be accepted and you can't write a comment.

10.3.1.1 Name

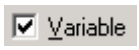
Each class element (function or variable) must have a unique name.
A name can be constructed of the alphanumeric characters and the underscore, but the latter may not be at in first place of the name.

Examples: Factorial, m_Table1

Each new name must differ from all other names of tokens, productions, functions and variables by at least one character.

10.3.1.2 Type

Depending on the activation of the variable check box in the toolbar of an ielement script



the type in the type field is the type of a class variable or the return type of a member function. In the first case a type has to be specified necessarily, in the second the type "void" is assumed, if no other type is specified.

10.3.1.3 Parameter

Depending on the activation of the variable check box in the toolbar of an interpreter script

the parameter field is invisible or visible.
For the syntax the same essentially applies, what is said for the parameters of productions.

Implicit xState parameter

A problem exists, if a call to a method shall be valid as well for the interpreter as for the generated c++ code. In the declarations of methods of the generated parser class an xState parameter is inserted automatically at the first position. For example:

```
void Method(state_type& xState);
```

To make the use of the interpreter more comfortable, you need not write the *xState* parameter in a call of this method. The interpreter always can access the state of the parser:

```
{- Method( ); -}
```

In the interpreter this code will be parsed and the missing parameter is added automatically. But if the code is exported, it simply will be copied, so that this parameter must be set explicitly.

```
{_  Method(xState); _}
```

If the semantic code shall be valid internally and externally *xState* has to be set:

```
{= Method(xState); =}
```

Since version 0.9.8.1 the *xState* parameter is inserted into the generated code automatically, if the Only copy option is not set, and if the method is not called within a part of code, which only can be exported.

In the project options you can activate a warning, which will appear, if such a state parameter is missing.

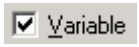
10.3.1.4 Comment

A comment to an interpreter script can be shown in the yellowish field. Temporary this field is also used to show error messages.

To change the comment, use the button. A dialog will be opened, where you can write the new text.

10.3.1.5 Text/Initialization

Depending on the activation of the variable check box in the toolbar of an interpreter script



the editor field is used for initialization code of the variable or for the body of a function.

a) **Initialization code**

Optionally for each class variable an initialization code can be written, which will be executed every time the parsing of a new input text begins.

Example:

Name: m_TotalPercent
Type: int
Init:

```
m_TotalPercent = 100;
```

Name: m_Numerals
Type: 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) **Function body**

The body of a function is written here.

Example:

Name: abs
Type: int
Parameter: int xi
Text:
```
if(xi < 0)
 xi = -xi;
return xi;
```

10.3.2 List of all instructions

Interpreted C++ instructions

Methods of the class 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)

Methods of the container classes

Mehtod of a vector

All cursor methods and additional

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)

Methods of a map

All cursor methods and additional

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)

Methods of the cursor class

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)

A map cursor has the additional methods

str **key**() const
bool **containsKey**(const str& xsKey) const
bool **findKey**(const str& xsKey)

Methods of a Function table

bool **add**(STRING, STRING);
void **visit**(const d/node& xNode)

Methods of the class 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**()

String manipulating functions

stod	to convert an str to double
stoi	to convert an str to int
hstoi	to convert a hexadicimal sting to int
stoc	to convert a stirng to a character
dtos	to convert a double value to an str
itos	to convert a int value to an str
itohs	to convert an interger into a hexadecimal string
ctohs	to convert a character into a hexadecimal string
ctos	to convert a character into a string

to_upper_copy	returns an upper case string
to_lower_copy	returns a lower case string
trim_left_copy	removes leading spaces
trim_right_copy	removes trailing spaces
trim_copy	removes leading and trailing spaces

File handling

basename	Returns the base name of a file path
extension	Returns the extension of a file path
change_extension	Changes the extension of a file pat
append_path	Composes a path
current_path	Returns the current path
exists	Tests the existence of a path
is_directory	Tests the existence of a directory
file_size	Returns the file size
find_file	Looks up a file in a directory
load_file	Loads a file
path_separator	String constant for the path separator

Output

out	writing the output
log	writing of log information
bool_bin	writes binary *bool*

int_bin	writes binary *int*
uint_bin	writes binary *unsigned int*
float_bin	writes binary *float*
double_bin	writes binary *double*
char_bin	writes binary *char*
string_bin	writes *char** with the length of the string
bin	writes passed type binary
ends	writes binary null character

Formatting instructions

unsigned int	**size**() const
str	**str**() const
void	**parse**(const str& xs)

Other functions

clock_sec	calculates time
time_stamp	date/time string
time_stamp(const str& xsFormat)	
random	generates random numbers
throw	throwing an exception
d/node	detach_node(const d/node& xn)

Parser class methods

Parser state

unsigned int	**size**() const
unsigned int	**length**(int sub = 0) const
str	**str**(int sub) const
bool	**matched**(int sub) const
bool	**matched**() 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 methods

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 Interpreted C++ instructions

The syntax of the interpreter is simple c++ syntax. All code, which works in the interpreter, can be compiled also by external c++ compilers and can be executed. This is not true the other way round. Tricky codings or code without consequences (e.g. returning a value, which will not be assigned) can cause errors in the interpreter with messages, hardly to understand.)

Tip.: Write simple code and better use two instructions than one.

10.3.3.1 C++

Some general remarks for programming newcomers:

C++ is a well-known very complex programming language. A simple sub set of this language, which is especially usefully to process text and which is understandable for programming newcomers, is integrated into the TextTransformer.

Generally a program consists in a sequence of instructions, which will be executed one after the other. Instructions mostly operate on data and data are represented by variables. For example: a variable with the name T could contain the text "tetra" and the instruction *to_upper_copy*(T) would transform this text into the upper case "TETRA".

C++ distinguishes different types of data, i.e. different types of variables represent different kinds of content. These types sometimes can be converted to each other, sometimes not. For example, a text is not a number. The attempt to convert the text "TETRA" into a number makes no sense. In a programming language, which don't use types such a conversion would be allowed and would have

unpredictable consequences. In this respect c++ is more save. The disadvantage is a little bit more work for the programmer: variables must be **declared** before they can be used.

10.3.3.2 Variable types

The TETRA interpreter knows elementary variable types, strings, nodes and container types and cursor types:

Type	Range of values	Default value (*)
bool	true/false (resp. 1/0)	false (resp. 0)
char	0-255	'\0'
int	-32768 - +32767	0
unsigned int	0 - 65565	0
double	-1.7E+308 - +1.7E+308 (15 digits)	0.0
str		""
node		node::npos
vector		empty
map		empty
cursor		–
function table		empty

* Default values are valid only inside of the interpreter. For the exported code, default values have to be set explicitely, if needed.

10.3.3.2.1 bool

Syntax

bool <identifier>;

Description

The key word **bool** indicates a data type, which only can have the values **false** or **true**. The key words **false** and **true** are boolean constants with predefined values. The numerical equivalent of **false** is null and **true** corresponds one.

A value of the type bool can be converted into a value of the type **int**. The numerical conversion sets **false** to null and **true** to one.

The other way round it is possible to convert **double** or **int** values into values of the type **bool**. Thereby an arithmetic null is converted to the value **false**, and each other value to **true**.

10.3.3.2.2 char

Syntax

char<variable_name>

Description

Use the type specifier char to define a character data type. Variables of type char are 1 byte in length.
Objects declared as characters (char) are large enough to store any member of the basic ASCII character set. The visual representation of certain nongraphic characters is possible by escape sequences.

10.3.3.2.3 int

Syntax

int <identifier> ;

Description

The **int** type specifier defines an integer data type, which can hold values in the range -32768 - +32767.

Integer constants can be written in an octal or hexadecimal notation too.

10.3.3.2.4 unsigned int

Syntax

unsigned int <identifier> ;

Description

The **unsigned int** type specifier defines an integer data type, which can hold values in the range 0 - +65565.

The **unsigned** type modifier designates, that the variable value will always be positive.

Integer constants can be written in an octal or hexadecimal notation too.

10.3.3.2.5 double

Syntax

double <identifier>

Description

The **double** type specifier defines a floating-point data type.

10.3.3.3 str

Syntax

str <identifier> ;

Description

str is the interpreter version of *std::string* or *std::wstring*. In the generated code **str** is defined by: *typedef std::string str* or, for unicode parser by: *typedef std::wstring str*.

A *str* variable contains chains of characters, that means: text. The control characters, which consist of the combination of a backslash with other characters also counts as a character, e.g. the linefeed '\n '. To use a backslash character in its literal meaning in a string, a second backslash has to be put in front of it: '\\'

Example:

```
str  s  =  "C:\\TextTransformer\\bin";
// wrong:   "C:\TextTransformer\bin"
```

The visual representation of other nongraphic characters is possible by escape sequences.

String literals adjacent to each other are unified into one single string literal. So you can better write longer texts.

Example:

```
str  s  =  "/*************************\n"
          "*  This is a comment       *\n"
           "*************************/\n";
```

In contrast to the previous variable types, *str* is not a basic type, but a derived type (a class), which has methods, by which you can get information about the contained text or by which you can manipulate it:

a) **Information**:

bool **empty**() const	returns true, if there is no text in the string
unsigned int **size**() const	returns the number of characters in the string
unsigned int **length**() const	returns the number of characters in the string

A whole family of methods is used to search for special positions inside of a string.

Parts of the string:

str **substr**(int from, int count) const
str **substr**(int from) const

returns the part of the string beginning at the character with the index *from* and of the length *count*

(or the rest of the text, if there aren't count characters any more). If there is no second parameter, all characters beginning at *from* will be returned.

Subscript-operator

Single characters of a string can be read by means of their index. The first character has index 0 and the last index size() - 1.

```
str s = "TextTransformer";
char c = s[5];  // now is: c == 'r'
```

It is not possible to change a character accessed by its index. Use the **replace**-method instead (see below)

b) **Manipulation**:

void **clear**() removes the text in the string
str& **replace**(unsigned int pos, unsigned int len, const str& s)

Replaces (at most) len characters starting with position pos with all characters of s. A reference of the string itself is returned.

Example:

```
str s = "wood";
out << s.replace(0, 1, "g") << " ";
out << s;

Output: good good
```

10.3.3.3.1 Searching

A group of methods can be used, to find certain positions inside of a string *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

Returns the index of the first occurrence of xs in the string, if there is one. Otherwise a special constant: *str::npos*, is returned. (*str::npos* only can be used in context of an equality operator.)

```
str s = "hello world";
unsigned int pos = s.find("o world");
if(pos != str::npos)
  out << s.substr(0, pos);
// else don't use value of pos
```

Prints: hell

You can pass the position, from where the search shall start, as a second parameter to the **find** -function.

```
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);   // prints : C:\Programme\TextTransformer\Target\
```

unsigned int **rfind**(const str& xs) const
unsigned int **rfind**(const str& xs, unsigned int pos) const

The *rfind* methods are working analogously to the according *find* methods, but *rfind* searches backward.
The result of the last example will be found faster by the *rfind* method:

```
str  s  =  "C:\\Programme\\TextTransformer\\Target\\test.txt";
unsigned int pos = s.rfind("\\");
if(pos != str::npos)
  out << s.substr(0, pos + 1);   // prints : 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

These functions search for characters of the string argument xs. By the optional second argument a position can be set, where the search shall start.

find_first_of
searches the first character of *xs*

find_first_not_of

searches the first character, which is not contained in *xs*

find_last_of
searches backwards the first character of *xs*

find_last_not_of
searches backwards the first character, which is not contained in *xs*

Example:

The following code inserts backslashes before single backslashes, the line feed and before the carriage return character. With xs = "Hello\r\nGood bye", you will get the result: "\"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

Containers contain and manage collections of elementary variables.
TETRA knows two basic types of containers and an adapted type:

vector
map
stack

You can iterate over the elements of a container or you can search for special elements. For this purpose serves a

cursor

Every container contains internally a cursor and the methods of the cursor are invoked directly as methods of the container. However, a cursor can be declared also externally.

The names of the container and cursor types are formed from an expression for the kind of the container and the short expression of the contained variable types.

type	vector	map	function table
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

A very special kind of container for the evaluation of parsing trees is a

function table

There isn't any cursor for function tables.

10.3.3.4.1 vector

A container class vector manages a list of elements. All elements of a list have the same data type. There are, however, different vector types depending on type of the contained elements. (See the table above)

Syntax

vbool <identifier>
vint <identifier>
vuint <identifier>
vdbl <identifier>
vchar <identifier>
vstr <identifier>
vnode <identifier>

Description

The vector types are ***typedefs*** of the clsss CTT_Vector< value_type > in the exported code. CTT_Vector is derived from std::vector< value_type >.

All vector classes have the same methods. These are

1.) the read only methods of the general cursor class:

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

By this method an external cursor is connected with the vector.

Example:

```
vint v;
vint::cursor cr =  v.getCursor();
```

cr now points to the same data element as the internal cursor, but further both can be placed independently from each other.

3.) Methods, which change the content of a vector:

void **reset**()

Positions the cursor before/behind the list of elements of the vector.

void **clear**()

removes all elements from the vector and resets all cursors.

void **push_back**(const value_type& xValue)

Appends xValue at the end of the vector. All cursors are reset.

void **pop_back**()

Removes the last element from the vector. All cursors are reset. If the vector is empty, an error occurs..You should check, whether the vector contains elements before calling *pop_back*:

bool **remove**()

removes the current element from the vector and returns *true*. If there is no current element or if an error occurs, *false* will be returned. All cursors are reset.

bool **setValue**(const value_type& xValue)

changes the value of the current element and returns *true*. If there is no current element or if an error occurs, *false* will be returned.

4.) Direct access of the elements

value_type **back**()

returns the last element. If the vector is empty, an error occurs..You should check, whether the vector contains elements before calling *back*:

```
if(v.size())
  value = v.back();
```

value_type **front**()

returns the first element. If the vector is empty, an error occurs..You should check, whether the vector contains elements before calling *front*:

```
if(v.size())
  value = v.front();
```

Index operator

The elements, which were inserted into a vector by the **push_back** method can be accessed directly by their index. The first element has index 0 and the last has index size() -1. If a vstr named v exists

```
str s = v[0];
```

copies the first String into *s* and

```
v[0] = s;
```

copies the string s into the first element, if the first element exists. If no first element was inserted by **push_back**, an error occurres.

10.3.3.4.1.1 Stack

There isn't a general stack container class of his own in the TextTransformer interpreter. Stacks can nevertheless be realized easily.

At first there are two special stacks for text-scopes and indentations. A stack also arises from

productions called recursively automatically (see below).

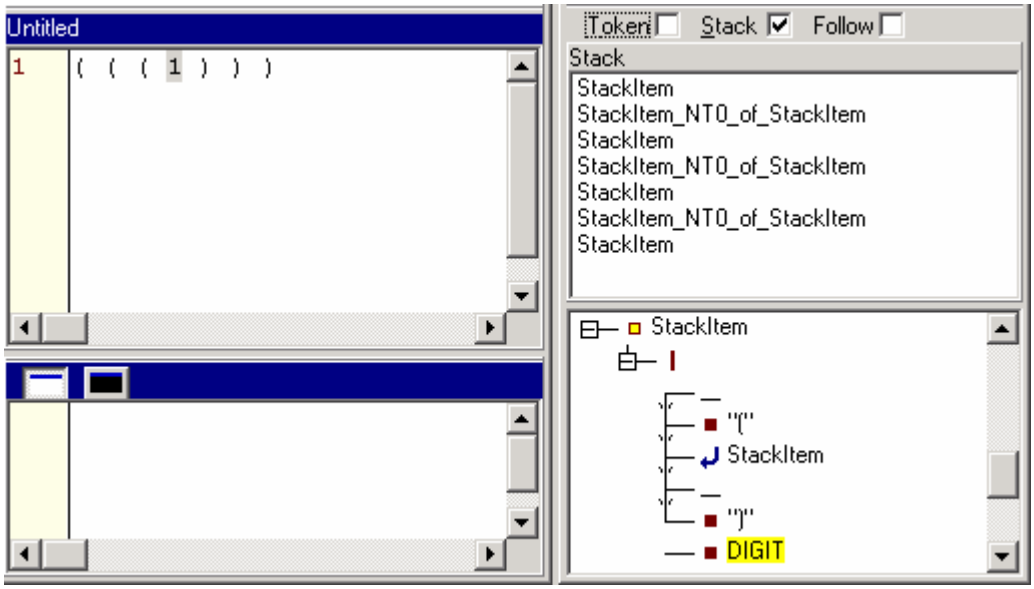
In other cases a vector can be used as a stack. A new value can be pushed on the stack with *push_back*, then the value can be accessed by *back* and finally it can be removed again by *pop_back*.

```
vint v;

for(int i = 1; i <= 3; i++)
   v.push_back(i);

while(v.size())
{
  out << v.back();
  v.pop_back();
}

// result: 321
```

E.g. a Stack arises automatically, when the text : "(((1)))", is parsed with the following start rule:

```
StackItem(int xi)

{{ int i = xi + 1; }}
"("
StackItem[i]
{{ out << i << endl; }}
")"
| DIGIT
```

For every new instance of stack a local variables i is created: i is pushed on the stack. When the stack production is left, the variable is destoyed: i is popped from the stack.



10.3.3.4.2 map

A container class map manages a list of pairs of values. All pairs of a list have the same data types. There are, however, different map types depending on type of the contained pairs. (See the table above)

Syntax

mstrbool <identifier>
mstrint <identifier>
mstruint <identifier>
mstrdbl <identifier>
mstrchar <identifier>
mstrstr <identifier>
mstrnode <bezeichner>
mstrdnode <identifier>

Description

The map types are ***typedefs*** of the class CTT_Map< value_type > in the exported code. CTT_Map is derived from std::map< value_type >.

A *map* variable is a list of pairs of values. Each pair consists of a key word and its value. The pairs are sorted alphabetically according to the key. Thus a map has a good performance, when searching for a key, but not so good performance when searching for a value.
A map is always in a certain state: one of the pairs is the current pair. This pair is denoted by the **cursor** (pointer). The cursor also can be invalid: it can be positioned before or behind the elements (pairs). In this case the property **hasCurrent** is *false* and to access the key or value makes no sense. The cursor can be positioned and the properties of the current pair of strings can read or manipulated.

All map classes have the same methods. These are

1.) the read only methods of the general cursor class:

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

By this method an external cursor is connected with the map.

Example:

```
mstrnode m;
mstrnode::cursor cr =  m.getCursor();
```

cr now points to the same data element as the internal cursor, but further both can be placed independently from each other.

3. In addition to these general methods the map as well as an external cursor of a map has the read only methods:

str **key**() const

returns the key of the current element or an empty string, if there is no current pair.

bool **containsKey**(const str& xsKey) const

returns *true*, if a key xsKey is contained in the map.

bool **findKey**(const str& xsKey)

searches for the key xsKey. If the key is contained in the map the according element becomes the actual element and the function returns *true*. Otherwise *false* will be returned.

4.) Methods, which change the content of a map:

void **reset**()

Positions the cursor before/behind the list of elements of the map.

void **clear**()

removes all elements and resets all cursors.

bool **add**(const str& key, const value_type& xValue)

inserts an element into the map with the key *key* and the value *value*. If the insertion succeeded, *true* is returned. If there was already a pair with the key *key* inside of the map, *false* will be returned and the value will not be overwritten. All cursors are reset.

bool **remove**()

removes the current element from the map and returns *true*. If there is no current element or if an error occurs, *false* will be returned. All cursors are reset.

bool **remove**(const str& xsKey)

removes the element with the key xsKey from the map and returns *true*. If the key isn't contained in the map or an error occures, *false* will be returned. All cursors are reset.

bool **setValue**(const value_type& xValue)

changes the value of the current element and returns *true*. If there is no current element or if an error occurs, *false* will be returned.

5.) Direct element access

If you have a **mstrstr** named m, you can insert elements or access their values directly by

```
m[key]
```

This either returns the value of the element with the key *key*, or inserts an element with *key*, if it does not yet exist. So by

```
m["one"] = "two";
```

you insert a value with the key "one" and the value "two" and by

```
str s = m["one"];
```

you can copy the value "two" into the string *s*. If a new element is inserted, all cursors are reset.

Attention: if there was not inserted a value for the key before, the latter statement will automatically insert an empty string as value for the key "one" and copy the empty string to s. If you want to test,

if the key exists in the map, you have to use the **findKey** function above.

10.3.3.4.3 cursor

A corresponding cursor type belongs to every container type. By means of the cursor single elements of the container can be searched for and the contents of the element can be read. Every container contains internally a cursor and the methods of the cursor are invoked directly as methods of the container. However, a cursor can be declared also externally. The **type name of the cursor** arises from the name of the accompanying container, by appending the expression ":: :cursor".

Example:

Use of an internal cursor:

```
vstr v;
v.push_back("tt");
v.push_back("TETRA");
v.reset();
while ( v.gotoNext() )
  out << v.value() << endl;
```

or declaration of an external cursor:

```
vstr v;
v.push_back("tt");
v.push_back("TETRA");
vstr::cursor cr = v.getCursor();
while ( cr.gotoNext() )
  out << cr.value() << endl;
```

The direct call of the internal cursor will be usually preferred to of the declaration of an external cursor because of its simplicity. However, if for example a range of the container shall be marked, then an external cursor is necessary.
The use of the internal cursor also has a side effect. By the change of the position of the internal cursor the state of the container is changed since the position of its cursor is part of this state. The internal cursor cannot be used therefore in a const parser.

The cursor itself can be in **three states**:

1. it is placed before the first (the same as behind the last) element: the function *hasCurrent* then returns *false*.

cr.hasCurrent() == false

2. it denotes a value. Now is

cr.hasCurrent() == true

If the content of the containers is changed by addition or removal of an element, the position of the cursor (not the cursor itself) gets invalid again:

cr.hasCurrent() == false

3. it is invalid, if there is no connection to a container:

cr.isValid() == false

A connection to a container is made by the call of the container method *getCursor* (see example above). But the connection can get lost, if the container stops existing.

Example:

```
vstr::cursor GetInvalidCursor( )
{
  vstr v;
  return vstr::cursor(v);
}
```

To call a method of an invalid cursor will not create an error, but will be useless.
Different to an external cursor an internal cursor never is invalid; its existence ends with the existence of the container.

10.3.3.4.3.1 General cursor methods

Cursors have read only access to the connected container. The cursors of different kinds of containers (vector and map) have a set of common methods, which will now be listed and explained.. Thereby an **element** is either a single value contained in a vector or a pair of values contained in a map.

a) Information:

bool **isValid**() const

returns, whether the cursor is connected to a container. For an internal cursor this is always the case.

bool **hasCurrent**() const

returns *true*, if the cursor is positioned on an actual element. Otherwise *false* will be returned.

bool **empty**() const

returns true, if there is no element in the container. If the container is invalid, *true* will be returned.

unsigned int **size**() const

returns the number of elements in the container. If the container is invalid, *0* will be returned.

value_type **value**() const

returns the value of the current element. If there is no current element, the default value of the value type will be returned: an empty string for the string type, 0 for numerical types and *false* for the bool type.

If the vaue type is *node*, you can call value().label() or value().value() to get the label or value of the node.

bool **gotoNext**()

sets the cursor on the next element. So it becomes the current element. If the current element was the last element of the list, *gotoNext* returns *false* and the cursor position becomes invalid. If the cursor position was invalid, it will be set on the first element by *gotoNext*.

bool **gotoPrev**()

sets the cursor on the previous element. So it becomes the current element. If the current element was the first element of the list, *gotoPrev* returns false and the cursor position becomes invalid. If the cursor position was invalid, it will be set on the last element by *gotoPrev*.

bool **containsValue**(const value_type& xValue) const

returns true, if a value xValue is contained in the container.

bool **findValue**(const value_type& xValue)

searches for the first value xValue. If the value is contained in the map the according element becomes the actual element and the function returns *true*. Otherwise *false* will be returned.

Example:

```
vstr v;
vstr::cursor cr = v.getCursor();
if ( ! cr.findNext("tt") )
  out << "v is empty";
```

bool **findNextValue**(const value_type& xValue)

Beginning at the actual position this function looks for a following element, the value which of is xValue. If such an element exists, it becomes the actual one and *true* is returned. Otherwise the actual position remains unchanged and *false* is returned.

bool **findPrevValue**(const value_type& xValue)

Beginning at the actual position this function looks for a previous element, the value which of is

xValue. If such an element exists, it becomes the actual one and *true* is returned. Otherwise the actual position remains unchanged and *false* is returned.

A map cursor has some additional methods:

str **key**() const
bool **containsKey**(const str& xsKey) const
bool **findKey**(const str& xsKey)

10.3.3.4.4 Function table

A function table is a map in which functions are sorted according to keys of strings. All functions of a list have the same type. There are, however, different function table types, depending on the type the functions return. (See the table above)

Syntax

mstrfun <identifier> ;
bool_mstrfun <identifier> ;
int_mstrfun <identifier> ;
uint_mstrfun <identifier> ;
dbl_mstrfun <identifier> ;
str_mstrfun <identifier> ;
node_mstrfun <identifier> ;
dnode_mstrfun <bezeichner> ;

Description

By means of these special containers the processing of a parse-tree is easier. To each label of a node a function can be assigned, which evaluates the node.
A function table shall be exemplified by *mstrfun*. *mstrfun* is similar to a map *mstrstr*: *mstrfun,* as it has an str key and another *str* as value. But the value here denotes the name of a class method. Class methods are added to a function table by their names. All methods must have the **same type**. The type of a function results from its return type and its parameters. So all functions contained in a table must have the same return type and the same number and kind of parameters. An additional condition is, that the **first Parameter** has to be the type: ***const node&***. By this parameter the node is passed, which shall be evaluated.
If you define an mstrfun-variable as an element of the TextTransformer, the **parameter field** is shown after you have specified the mstrfun-type in the field for the return type. Here you have to specify the parameters in the same manner as usual for the functions.
The special type of the function table determines the **return type** of the functions contained in the table:

type of function table	**return type**
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);

In the text field you now can insert the functions by means of the *add*-command.
All functions, which *mstrfun* shall contain, must be inserted here. It is not possible to create *function-tables* dynamically, while a transformation is running.

LABEL and FUNCTION can be passed either as strings - e.g. "number" - or as identifiers - e.g. *number*. To pass a parameter as an identifier has the advantage that the syntax highlighting then is active and you can change to the according function by a mouse click.
When C++ code is created for the export, the identifiers will be translated to strings or functions addresses automatically.
It is not possible, to use general expressions for these parameters.

Example:

m_Eval might be an mstrfun, of void functions with only one *node* parameter, like

void VisitVariable(const node& xNode)

The first and the second of the following calls are correct, the third not:

```
m_Eval.add( "Variable", "VisitVariable");
m_Eval.add( Variable, VisitVariable);

str sLabel =  "Variable";
m_Eval.add( sLabel, "VisitVariable"); // error
```

Default function

Each function table must contain a default function, which will treat the nodes with labels, which are different from all key values in the table. The default function is inserted by the empty string "" as key:

Previous example continued:

```
void Default(const node& xNode)
m_Eval.add( "", "Default");
```

The extension of function tables by new functions is made easier by the Function-Table-Wizard

. **visit**(const node& xNode ...)

A function of the table is called by a call of the *visit* method of the table. Depending on the value of the label of the passed node, the call of *visit* will be redirected automatically to the function, which is assigned to the value of the label.

Previous example continued:

```
node n( "Variable", ...);
m_Eval.visit( n );
```

is equivalent with

```
if ( n.label() == "Variable")
    VisitVariable( n );
 else
   Default( n );
```

In the exported c++ code mstrfun is implemented as CTT_Mstrfun.

10.3.3.5 node / dnode

Syntax

node <identifier> ;
dnode <identifier> ;

Description

The types *node* and *dnode* are designating the structure of a tree node. Such a node consists in a first string - the label - to identify the kind of the node and a second string, which contains a value. Finally, an unsigned int value can be assigned to the node. A special property of nodes is, that they can be combined to a common tree structure.

node and *dnode* nearly have the same interface and therefore can be used analogously. The difference is discussed below.

According to these properties *nodes* can be used in the TextTransformer:

- as a container to store single data
- as parse tree to represent the grammatical structure of the whole input text

The names and relations of the nodes are explained in the glossary.

Node-instances have special properties: they are reference counted pointers. If a node is assigned to a different, a change of the value (or label) of one of these nodes will result in a change of the value (or label) of the other too.

Example.:

```
node n1("label1", "value1");
node n2("label2", "value2");
n1 = n2;
n1.setLabel("label3");
// now as well n1 as n2 have the label "label3" and the value "value2"
```

A tree exists as long as there exists a reference to one of its nodes.

The *node* function are listed into the following chapters:

Construction
Information
Neighbors
Search
dnode specials

10.3.3.5.1 node: Construction

You can create single nodes with initializing parameters or without:

node()
node(str& xsLabel)
node(str& xsLabel, str& xsValue)

A new node object will be created with a label and a value according to the parameters.

node(**const node**& xOther)

A **reference** on the node object xOther will be created. That means, the new node has the same label, value and the same children as the old. If one of the nodes will be changed, the other will change too.

A copy (isolated form the tree) of a node is yielded by:

node clone()

Example:

```
node  nCopy  =  root_node.bottomLastChild().clone();
```

To add a node to a tree structure use one of the following functions:

bool addChildFirst(**const node**& xNewChild)
bool addChildLast(**const node**& xNewChild)
node add(**const str**& xsLabel, **const str**& xsValue)

By the first function the new node becomes the first child node of the node, of which you called *addChildFirst*. By the call of *addChildLast* the passed node will become the last child. *true* is returned, if the insertion was successful.
The third function *add* can be read as an abbreviation of:

```
addChildLast( node( xsLabel, xsValue )
```

that means a new node is created with the label xsLabel and the value xsValue and inserted as the last child node. The new node is returned or node::npos, if the insertion was not successful.
If the *xNewChild* is already in a tree, it is first removed there.
If *xNewChild* has children, they are transported too.

bool addChildBefore(**const node**& xnNewChild, **const node**& xnRefChild)

Inserts the node *xnNewChild* before the existing child node *xnRefChild*.
If refChild is null, insert newChild at the end of the list of children.

node removeChild(**const node**& xnOldChild)

Removes the child node indicated by xnOldChild from the list of children, and returns it. If *xnOldChild* is not a child *node::npos* is returned.

node detach_node(**const node**& xn)

This global function simplifies the use of *removeChild*, because it has not to be called with the parent node. The function is defined as:

```
if(xn.parent()  != node::npos)
    xn.parent().removeChild(xn);
return xn;
```

bool replaceChild(**node**& xNewChild, **node**& xOldChild)

Replaces the child node *xOldChild* with *xNewChild* in the list of children, and returns the success.

***node* and *dnode* behave differently if it is tried, to insert a d/node, which was already inserted in a tree, at another position:**

Up to version 1.7.2 this was **forbidden** for a **node** and you got an error message. From version 1.7.3 on, the behavior is assimilated to the behavior of **dnode**'s

For a ***dnode*** this operation is a **movement**: the *dnode* with its sub-nodes is inserted in the desired place, however, disappears at the old position. This can be quite useful.

Trees from *node's* and *dnode*'s have a different memory management: *node's* are reference-counted internally and *dnode's* are managed by the XMLDocument externally. The renunciation of such a "factory" has the consequence of more effort in case of the removal of a node from a tree.

10.3.3.5.2 node: Information

Information about the data of a node itself or about its position inside of a tree can be obtained by the following functions:

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)
```

str **label**() const

Returns the value of the label.

void **setLabel**(const str& xsLabel)

Sets the label of the node..

str **value**() const

Returns the value of the node.

void **setValue**(const str& xsValue)

Sets the value of the node.

unsigned int **id**() const
void setId(unsigned int xuiId)

These functions are reserved for further developments of the TextTransformers. The id type may change in future.

bool **hasChildren**() const

The function returns *true*, if the node has a child node; otherwise false will be returned.

bool **isDescendant**(const node& xNode) const

By *isDescendant* you can determine, whether the passed node *xNode* contains to the branch, which has its origin in the actual node.

bool **isAncestor**(const node& xNode) const

By *isAncestor* you can determine, whether the actual nodes contains to the branch, which has its origin in the passed node *xNode*.

bool **isSibling**(const node& xNode) const

By *isSibling* you can determine, whether the passed node *xNode* follows in the sequence of siblings of the actual node.

unsigned int **level**() const

This function returns 0 for the root node; the first child of the root has the level 1 and so on

unsigned int **descendentsCount**() const

This function returns the number of nodes in the branch, which has its origin in the actual node.

unsigned int **childCount**() const

This function returns the number of nodes, which immediately are subordinated to the actual node.

Attributes

Attributes consist of a key and an accompanying value. E.g. attributes are useful for the storage of ini entries: while the sections of the ini file are represented by nodes, the values of the section can be put as attributes.

void **setAttrib**(const str& xsLabel, const str& xsValue)

For every node arbitrarily many attributes can be set with *setAttrib*. An attribute has a name *xsLabel* and a value belonging to it, *xsValue*. If the name isn't existing in the list of the attributes yet, then new attribute is inserted. If the name already exists, the according value will be overwritten with the new value.

str **attrib**(const str& xsLabel)

By *attrib* the value with the name *xsLabel* can be read.

bool **hasAttrib**() const

Returns *true*, if the node has at least one attribute. Otherwise *false* is returned.

Example:

a) Parsing the ini file:

```
{{node n = xIni.add("[FONT]", ""); }}

"NAME"  "=" Value    {{ n.setAttrib("NAME", State.lp_str()); }}
"COLOR" "=" Value    {{ n.setAttrib("COLOR", State.lp_str()); }}
```

b) Using the values:

```
node n = Ini.findNextLabel("[FONT]")

SetValues(n.attrib("NAME"), n.attrib("COLOR"));
```

10.3.3.5.3 node::npos

A special node is: *node::npos.* It is similar to str::npos or to a NULL-pointer in c++. All functions, which return a *node* object, will return *node::npos,* if the node doesn't exist. For example a just created node will neither have child nodes nor a parent:

```
node nNew;
node nPos = nNew.firstChild();
// now is: nPos == node::npos
```

Before you do something with a node, which is the result of a neighbor- or search-function, you should always compare it first with node::npos:

```
if(nPos != node::npos)
{
  // do something with nPos
}
else
 // do nothing with nPos
```

All node functions, which return a different node, applied to *node::npos*, will return *node::npos* again. *node::npos* cannot be inserted into a tree.

10.3.3.5.4 node: Neighbors

By the following functions you can get nodes, which are in certain positions relatively to another node:

```
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

If you know the names and relations of the nodes as explained in the glossary, the names of most of these functions are self-explanatory. Each function returns the neighbor node according the name of the function.

Example:

```
{{
  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();
  }
}}

results in:

label: label_11, value: value_11
label: label_12, value: value_12
```

After the execution of this assignment nChild is equal to the first child node of the node nParent. If nParent has no child, the following is valid

nChild == node::npos

Further explanation is needed for:

node bottomFirstChild() const

First child of the first child ...

node bottomLastChild() const

Bottom last child of the last child

node next() const
node prev() const

next returns the next node in descending direction. *prev* teturns the next node in ascending direction
.

node follow() const

follow returns the next node in descending direction, which follows on the last child node. This is either the *nextSibling* of the actual node or the *nextSibling* of the first of the *parent* nodes, which has a *nextSibling* or node::npos.

node nextLeaf() const
node prevLeaf() const

next returns the next node in descending direction, which has no child nodes. *prev* teturns the next node in ascending direction, which has no child nodes.

10.3.3.5.5 node: Searching

Starting from a node other nodes with certain data can be searched:

```
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

Returns the next node in descending direction, of which the label is equal to the passed string xsLabel. If no such node exists, the function returns node::npos.

With the second optional parameter *xnLast* a tree node can be chosen at which the search is finished.

Example:

```
node pos = xn.findNextLabel( "parameter" );

while( pos != node::npos )
{
  out << "parameter: "  << pos.value() << endl;
   pos = pos.findNextLabel( "parameter" );
}
```

This function outputs all parameters of all possible parameter lists which follow on xn.
If only the parameters of a current list shall be written, the code can be changed to:

```
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

Returns the next node in descending direction, of which the value is equal to the passed string xsValue. If no such node exists, the function returns node::npos.
With the second optional parameter *xnLast* a tree node can be chosen at which the search is finished.

node findPrevLabel(const str& xsLabel) const
node findPrevLabel(const str& xsLabel, const node& xnLast) const

Returns the next node in ascending direction, of which the label is equal to the passed string xsLabel. If no such node exists, the function returns node::npos.
With the second optional parameter *xnLast* a tree node can be chosen at which the search is finished.

node findPrevValue(const str& xsValue) const
node findPrevValue(const str& xsValue, const node& xnLast) const

Returns the next node in ascending direction, of which the value is equal to the passed string xsValue. If no such node exists, the function returns node::npos.
With the second optional parameter *xnLast* a tree node can be chosen at which the search is finished.

node findChildLabel(const str& xsLabel, bool xbRecursive = true)

Looks for the next child node, the label which of is *xsLabel*. If such a node exists it is returned, otherwise node::npos is returned.

If *xbRecursive* is *true*, the label is looked up in the child nodes of the children too.

node findChildValue(const str& xsValue, bool xbRecursive = true)

Looks for the next child node, the value which of is xsValue. If such a node exists it is returned, otherwise node::npos is returned.
If *xbRecursive* is *true*, the label is looked up in the child nodes of the children too.

node findParentLabel(const str& xsLabel)

Looks for the next node in the set of parent nodes, the label which of is xsLabel. If such a node exists it is returned, otherwise node::npos is returned.

node findParentValue(const str& xsValue)

Looks for the next node in the set of parent nodes, the value which of is xsValue. If such a node exists it is returned, otherwise node::npos is returned.

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)

Analogously to the aforementioned functions you can search with these functions for a certain ID value.

10.3.3.5.6 node: Sorting

void sortCildrentA()
void sortCildrenD()

By calling these methods the direct children of the node are sorted. *sortCildrentA* sorts the nodes according to their labels in alphabetical ascending order and *sortCildrenD* sorts in alphabetical descending order.

10.3.3.5.7 dnode specials

node and *dnode* nearly have the same interface and therefore can be used analogously. In contrast to *node, dnode* is part of a xerces document. Such documents open a big room for to the manipulation in the produced C++ code and they can easily be written as XML documents. Trees from *dnodes* as opposed to such from *nodes* are also shown in the variable-inspector after a transformation is ended.This lies in the fact, that the document is part plugin which exists outside the parser while nodes exist only locally within the parser. This is also the reason why dnode class elements cannot be initialized before the beginning of parsing. Only then the plugin is passed into the parser so that the *dnodes* can be produced by the DOMDocument of the plugin. At the

generation of C++ code which uses dnodes the xerces library must be linked. In the project options for the code production the CTT_ParseStateDomPlugin has to be set to as a plugin type then. The evaluation by means of function tables in the exported code isn't implemented yet.

The root node of a dnode tree must be produced with the function GetDocumentElement. The document finally can be written with the function WriteDocument

The way how the document is written is set in the project options.

While label and value of a *node* can consist of arbitrary characters, the set of the **characters for the the label a *dnode* are restricted** on letters, numbers and the underline. The label may not start with a number or the underline. Special characters aren't permitted here. Exactly the characters are permitted, that are also are permitted for the definition of an XML-tag.

dnode's can be moved in a tree and *nodes* cannot. This difference was already metionned above.

10.3.3.6 const

Syntax

const <variable name> [= <value>] ;

<function name> (const <type>*<variable name> ;)

<function name> const;

Description

Use the const modifier to make a variable value not modifiable.
Use the const modifier to assign an initial value to a variable that cannot be changed by the program. Any future assignments to a const result in a compiler error.

10.3.3.7 Operators

Operators can be applied to the elementary data types and strings and partly also to other types:

Arithmetic operators
Assignment operators
Relational operators
Equality operators
Logical operators

10.3.3.7.1 Arithmetic operators

Following arithmetic operators exist:
Binary operators:

Op1 * Op2
Op1 / Op2

Op1 % Op2 (modulus or remainder)
Op1 + Op2
Op1 - Op2

Unary operators:
Op++
Op1--

Applied on types, which represents numbers, the operators + (addition), - (subtraction), * (multiplication) and / (division) execute the according basic arithmetical operation.
(op1 % op2) Remainder of (op1 divided by op2)

For / and %, op2 must be nonzero op2 = 0 results in an error. (You can't divide by zero.)
% cannot be applied on the double type.

The operator ++ (increment) adds the number 1 to the value of the expression.
The operator -- (decrement) subtracts the number 1 of the value of the expression.

Applied on strings the addition causes a concatenation of the characters.

10.3.3.7.2 Assignment operators

Following assignment operators exist:

= *= /= += -= %= ^= |= &= <<= >>=

The value of the operand Op1 after execution of the assignment

Op1 = Op2;

is equal to the value of Op2.

The expression

Op1 op= Op2;

has the same effect as

Op1 = Op1 op Op2;

Example: Op1 += Op2; is equivalent to Op1 = Op1 + Op2;.

The operands Op1 and Op2 must be either of the same type or they must be compatible to each other.

For Op2 also the call of a production or a token can be substituted, if they return a compatible value, and if the closing bracket of the semantic action is immediately following the operator:

Example:

```
{{e = }} Term
```

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

10.3.3.7.3 Relational operators

Syntax

relational-expression < shift-expression

relational-expression > shift-expression
relational-expression <= shift-expression

relational-expression >= shift-expression

Remarks

Use relational operators to test equality or inequality of expressions. If the statement evaluates to be true it returns a nonzero character; otherwise it returns false (0).

> greater than
< less than
>= greater than or equal
<= less than or equal

In the expression

E1 <operator> E2

the operands must be both of arithmetic type.

10.3.3.7.4 Equality operators

There are two equality operators: == and !=. They test for equality and inequality between arithmetic values or strings, following rules very similar to those for the relational operators.

Note:
Notice that == and != have a lower precedence than the relational operators < and >, <=, and >=. Also, == and != can compare the string type for equality and inequality where the relational operators would not be allowed.

The **syntax** is

equality-expression:

relational-expression
equality-expression == relational-expression

equality-expression != relational-expression

10.3.3.7.5 Logical operators

Syntax

logical-AND-expr && inclusive-OR-expression

logical-OR-expr || logical-AND-expression

! cast-expression

Remarks

Operands in a logical expression must be of scalar type.

- **&&** logical AND; returns true only if both expressions evaluate to be nonzero, otherwise returns false. If the first expression evaluates to false, the second expression is not evaluated.
- **||** logical OR; returns true if either of the expressions evaluate to be nonzero, otherwise returns false. If the first expression evaluates to true, the second expression is not evaluated.
- **!** logical negation; returns true if the entire expression evaluates to be nonzero, otherwise returns false. The expression !E is equivalent to (0 == E).

10.3.3.7.6 Bitwise operators

Remark

Use the bitwise operators to modify the individual bits rather than the number.

Operator	What it does
&	bitwise AND; compares two bits and generates a 1 result if both bits are 1, otherwise it returns 0.
|	bitwise inclusive OR; compares two bits and generates a 1 result if either or both bits are 1, otherwise it returns 0.
^	bitwise exclusive OR; compares two bits and generates a 1 result if the bits are complementary, otherwise it returns 0.
~	bitwise complement; inverts each bit. ~ is used to create destructors.
>>	bitwise shift right; moves the bits to the right, discards the far right bit and assigns the left most bit to 0.
<<	bitwise shift left; moves the bits to the left, it discards the far left bit and assigns the right most bit to 0.

Both operands in a bitwise expression must be of an integral type.

Bit value E1	Bit value E2	Result of E1 & E2	Result of E1 ^ E2	Result of E1 | E2
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

Note: &, >>, << are context sensitive:
& can also be the reference operator.
>> can also be the input operator in I/O expressions.
<< can also be the output operator in I/O expressions.

10.3.3.7.7 Conditional operator

Syntax

logical-OR-expr ? expr : conditional-expr

Remarks

The conditional operator ?: is a ternary operator.
In the expression E1 ? E2 : E3, E1 evaluates first. If its value is true, then E2 evaluates and E3 is ignored. If E1 evaluates to false, then E3 evaluates and E2 is ignored.
The result of E1 ? E2 : E3 will be the value of either E2 or E3 depending upon which evaluates.

The condition operator only is in the TETRA interpreter of a very restricted use since for the E's not the full spectrum of c++ expressions is permitted at present. You can get more exact information about this by looking into the parser for the TETRA c++-interpreter.

10.3.3.8 Control structures

Following control structures can be used inside of the TETRA-interpreter

if, else
for
while
do
switch

10.3.3.8.1 if, else

Syntax

if (<condition>) <statement1>;

if (<condition>) <statement1>;
else <statement2>;

Description

Use if to implement a conditional statement.
The condition statement must convert to a bool type. Otherwise, the condition is ill formed.

When <condition> evaluates to true, <statement1> executes.
If <condition> is false, <statement2> executes.
The else keyword is optional, but no statements can come between an if statement and an else.

In contrast to the C/C++ standard inside of the condition no variables may be defined and no assignments can be executed. That means the following is not possible:

```
if (int val = stoi("1")) or  // false
int val; if (val = stoi("1"))   // false
```

Possible is:

```
if (stoi("1"))
```

10.3.3.8.2 for

Syntax

for ([<initialization>] ; [<condition>] ; [<increment>]) <statement>

Description

The **for** statement implements an iterative loop.
<condition> is checked before the first entry into the block.
<statement> is executed repeatedly UNTIL the value of <condition> is false.

Before the first iteration of the loop, <initialization> initializes variables for the loop.
After each iteration of the loop, <increments> increments a loop counter. Consequently, j++ is functionally the same as ++j.

In C++, <initialization> can be an expression or a declaration. The scope of any identifier declared within the for loop extends to the end of the control statement only. A variable defined in the for-initialization expression is in scope only within the for-block.

10.3.3.8.3 while

Syntax

while (<condition>) <statement>

Description

Use the **while** keyword to conditionally iterate a statement.
<statement> executes repeatedly until the value of <condition> is false.
If no condition is specified, the **while** clause is equivalent to **while(true)**. The test takes place before <statement> executes. Thus, if <condition> evaluates to false on the first pass, the loop does not execute.

10.3.3.8.4 do

Syntax

do <statement> while (<condition>);

Description

The **do** statement executes until the condition becomes false.
<statement> is executed repeatedly as long as the value of <condition> remains true.
Since the conditon tests after each the loop executes the <statement>, the loop will execute at least once.

10.3.3.8.5 switch

Syntax

```
switch ( <switch variable> ) {

  case <constant expression> : <statement>; [break;]
    .
    .
    .
  default : <statement>;

}
```

Description

Use the switch statement to pass control to a case, which matches the <switch variable>. At which point the statements following the matching case evaluate.
If no case satisfies the condition the default case evaluates.
To avoid evaluating any other cases and relinquish control from the switch, terminate each case with break;.

10.3.3.9 Output

The result of a transformation appears on the screen or in a file if it is written there. Representatives of these output destinations are

out	for the results of the transformation
log	for additional information about a transformation

Normally TextTransformer generates text files. It is also possible for special professional purposes to produce binary files.

10.3.3.9.1 out

out represents the target text: inside of the IDE working space this text appears in the target window. In the Transformation-Manager or the command line tool and the generated c++ parser, the target text is written into a file. To write a text or the content of a variable to *out*, the shift operator "<<" is used.

Example:

May *sResult* be a string variable, which holds the text "42", the following instruction:

```
out << sResult;
```

causes the appearance of "42" in the target window. Shift instructions can be chained:

```
out << "The result is: " << sResult << "\n";
```

This chain of instructions writes

"The result is: 42"

into the target window. (The character '\n' represents a line feed. When this character is written to *out*, the next output will be written into the next line.)

The result of a call of a production or a token, which returns a value, can be written to the output directly. For this, the closing bracket of the semantic action must follow the shift operator directly and the bracket must be followed by the call of the rule. Example:

```
{{ out << }} Rule
```

Remark 2: The output to *out* can be redirected into another file by RedirectOutput.

Remark 3: If c++ code is generated, *out* is replaced by the expression: *xState.out()*. *xState.out()* returns an ostream object from the plugin.

endl

```
out << endl
```

is another notation for

```
out << '\n'
```

In c++ this command also flushes the output buffer.

Remark: Up to the version 0.9.8.6 of the TextTransformer **cout** had to be written instead of *out*. *cout* represents the standard output channel, normally the console.

10.3.3.9.2 log

Similar, as an output can be directed to out, you also can direct an output to **log**. But while the first output is appended to the target text, the second will appear in the log window. This output can provide Meta information about the course of a program and can help to search possible faults.

Example:

```
log << "value: " << xs;
```

If the parser will be exported as c++ code, the *log* commands will not be written, if you havn't activated the option only to copy the code. To make it possible to generate executable programm code too, when this option is activated, you can write "clog" instead of "log".

10.3.3.9.3 Binary output

For writing variables into binary files there is a simple notation in the TextTransformer:

Example:

```
out << int_bin( 42 ) << double_bin( 123,456 );
```

or simpler:

```
out << bin( 42 ) << bin( 123.456 );
```

The second notation is translated into the first one automatically in the exported C++ code. In these expressions an overwritten output operator is called for temporary objects which provide the writing of the binary forms of the respective variable types.

The variable types bool, char, int, unsigned int, float and double can be written binarily in this way. string_bin writes string c_str into the output.

```
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 )
```

A special case is the zero character '\0'. To write this binarily there is this besides *char_bin*('\0') the manipulator *ends*.

```
out << ends; // writes 00
```

To prevent, that '\n' will be converted in "\r\n', the file should be opened in the binary mode. Binary data are represented in the output window of the TextTransformer only mutilated.

10.3.3.10 return

Syntax

return <expression> ;

Description

Use the return statement to exit from the current function back to the calling routine, returning a value.
As well productions, as element functions as token actions can return values.

Remark: the type of the returned value must be compatible with the type declared in the according field of the script.

10.3.3.11 break

Syntax

break ;

Description

Use the break statement within loops to pass control to the first statement following the innermost switch, for, while, or do block.

10.3.3.12 continue

Syntax

continue ;

Description

Use the continue statement within loops to pass control to the end of the innermost enclosing brace; at which point the loop continuation condition is re-evaluated.

10.3.3.13 throw

With an instruction of the kind:

throw CTT_Error("error message");

a TETRA program can be canceled. An arbitrary text or a string variable can be passed as message. In the C++ jargon for this statement is said, that the exception "CTT_Error" is thrown.

Inside of the TextTransformer, the text will be shown in the log window.

User defined error message: error message.

Remark: Such a break is executed in the term production of the calculator example to prevent a division by null.

An alternative to the *throw* command is the function *GenError*.
You can force a stop of a program also, if the interpreter is deactivated. For this the key word EXIT is defined.

10.3.4 String manipulation

Inside of the TextTransformer some instructions are available, which are no standard instructions.

stod	to convert an str to double
stoi	to convert an str to int
hstoi	to convert a hexadicimal sting to int
stoc	to convert a stirng to a character
dtos	to convert a double value to an str
itos	to convert a int value to a str
itohs	to convert an interger into a hexadecimal string
ctohs	to convert a character into a hexadecimalstring
ctos	to convert a character into a string

Remark: An integer variable, which represents an ANSI value, can be converted directly into the character by assignment to a char variable:

```
char c = 65; //  c == 'A'
```

to_upper_copy	returns an upper case string
to_lower_copy	returns a lower case string
trim_left_copy	removes leading spaces
trim_right_copy	removes trailing spaces
trim_copy	removes leading and trailing spaces

In the professional version the source code for these instructions is delivered in the file: tt_lib.cpp

10.3.4.1 stod

Prototype

double stod(const str& xs)

Description

Converts an str into a double number.

Return value

stod returns the result of the conversion or throws an exception (boost::bad_lexical_cast), if the string cannot be converted to a double value.

10.3.4.2 stoi

Prototype

```
int stoi(const str& xs)
```

Description

Converts an str into an integer.

Return value

stoi returns the result of the conversion or throws an exception (boost::bad_lexical_cast), if the string cannot be converted to an integer.

10.3.4.3 hstoi

Prototype

```
int hstoi(const str& xs)
```

Description

Converts an *str*, which consists of hexadicimal characters, into an integer.

Return value

hstoi returns the result of the conversion or 0, if the string cannot be converted to an integer.

Example:

The result of hstoi("FF") is the number 255.

10.3.4.4 stoc

Prototype

```
char stoc(const str& xs)
```

Description

Converts an *str* into a character.

Return value

Returns the first character of the string *xs* or, if the string is empty, '\0'.

Example:

```
char c = stoc("hello");
// c == 'h'
```

10.3.4.5 dtos

Prototype

str dtos(double xd)

Description

Converts a double value into an str.

Return value

dtos returns the result of the conversion or throws an exception (boost::bad_lexical_cast), if the value cannot be converted to a string.

10.3.4.6 itos

Prototype

str itos(int xi)

Description

Converts an integer into an *str*.

Return value

itos returns the result of the conversion or throws an exception (boost::bad_lexical_cast), if the value cannot be converted to a string.

10.3.4.7 itohs

Prototype

str itohs(int xi)

Description

Converts an integer into an *str*, which consists of hexadecimal characters.

Return value

itohs returns a string as result of the conversion

Example:

The result of *itohs*(1000) is the string: "3e8".

10.3.4.8 ctohs

Prototype

str ctohs(char xc)

Description

Converts a character into an *str*, which consists of hexadecimal characters.

Return value

ctohs returns a string as result of the conversion

Example:

The result of ctohs('A') is the string: "41".

10.3.4.9 ctos

Prototype

str ctos(char xc)

Description

Converts a character into an *str*.

Return value

returns a string, which contains *xc* as single character.

Example:

```
str s = "hello";
s = ctos( s[ 0 ] );
// s == "h";
```

10.3.4.10 to_upper_copy

Prototype

str to_upper_copy(const str& xs)

Description

Returns a string, which is created from the string xs, by converting each character to upper case.

Example:

```
str s = to_upper_copy("TextTransformer");
```

has the result: s == "TEXTTRANSFORMER".

10.3.4.11 to_lower_copy

Prototype

str to_lower_copy(const str& xs)

Description

Returns a string, which is created from the string xs, by converting each character to lower case.

Example:

```
str s = to_lower_copy("TextTransformer");
```

has the result: s == "texttransformer".

10.3.4.12 trim_left_copy

Prototype

str trim_left_copy(const str& xs)

Description

Remove all leading spaces from the input. The result is a trimmed copy of the input.

Example:

```
str s = trim_left_copy("   TextTransformer   ");
```

has the result: s == "TextTransformer   ".

10.3.4.13 trim_right_copy

Prototype

```
str trim_right_copy(const str& xs)
```

Description

Remove all trailing spaces from the input. The result is a trimmed copy of the input.

Example:

```
str s = trim_right_copy("   TextTransformer   ");
```

has the result: s == "   TextTransformer".

This command is useful, to extract text, which is covered by a SKIP symbol. When the parser recognizes the token after the SKIP symbol, leading spaces are ignored, but not the spaces before the token. For example:

SKIP {{ **out** << **trim_right_copy**(**xState**.str()); }} "$"

input : " 77.74 $"
output : "77.74"

xState.str() == "77.74 " and trim_right_copy(xState.str()) == "77.74".

10.3.4.14 trim_copy

Prototype

```
str trim_copy(const str& xs)
```

Description

Remove all leading and trailing spaces from the input. The result is a trimmed copy of the input

Example:

```
str s = trim_copy("   TextTransformer   ");
```

has the result: s == "TextTransformer".

10.3.5 File handling

The following functions for the path and file treatment are based on the portable boost filesystem library.

basename	Returns the base name of a file path
extension	Returns the extension of a file path
change_extension	Changes the extension of a file pat
append_path	Composes a path
current_path	Returns the current path
exists	Tests the existence of a path
is_directory	Tests the existence of a directory
file_size	Returns the file size
find_file	Looks up a file in a directory
load_file	Loads a file
load_file_binary	Loads a file in binary mode
path_separator	String constant for the path separator

see also: source and target, redirection, unit_dependence example

10.3.5.1 basename

Prototype

str basename(**const** str& xsPath);

Description

If the substring after the last path separator in the path xsPath contains a dot ('.'), the substring ending at the last dot (the dot is not included) is returned. Otherwise the entire substring is returned.

Example 1:

```
out << basename("D:\\TextTransformer\\Settings\\EditDefault.ds");
```

Output::

```
EditDefault
```

Example 2:

```
out << basename("D:\\TextTransformer\\Settings");
```

Output::

```
Settings
```

10.3.5.2 extension

Prototype

str extension(**const** str& xsPath);

Description

If the substring after the last path separator in the path xsPath contains a dot ('.'), the substring starting with the dot is returned. Otherwise an empty string is returned.

Example:

```
out << extension("D:\\TextTransformer\\Settings\\EditDefault.ds");
```

Output::

```
.ds
```

10.3.5.3 change_extension

Prototype

str change_extension(**const** str& xsPath, **const** str& xsNewExtension);

Description

This function returns the path, which results from changing the extension in the path xsPath to xsNewExtension. xsNewExtension should include a dot to achieve reasonable results.

Example 1:

```
out << change_extension("EditDefault.ds",   ".tb");
```

Output::

```
EditDefault.tb
```

Example 2:

```
out << change_extension("EditDefault.ds",  "tb");
```

Output::

```
EditDefaulttb   // presumably not wanted
```

10.3.5.4 append_path

Prototype

str append_path(**const** str& xsPath1, **const** str& xsPath2);

Description

Returns a path combined of the parts xsPath1 and xsPath2 in which they are connected by a path separator. If xsPath1 or xsPath2 is empty, then the respectively other part is returned without attaching a path separator.

Example:

```
str sLogPath = append_path ( current_path(),  "log.txt");
```

instead of

```
str sLog = current_path() + path_separator + "log.txt";
```

Under Windows is valid:

```
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

Prototype

str current_path();

Description

Returns the current path as maintained by the operating system.

10.3.5.6 exists

Prototype

bool exists(**const** str& xsPath);

Description

Returns *true,* if the operating system reports the path represented by xsPath exists, else *false*.

Example:

```
if(exists(TargetName())
  throw CTT_Error("target file already exixts");
```

10.3.5.7 is_directory

Prototype

bool is_directory(**const** str& xsPath);

Description

Returns *true*, if the operating system reports the path represented by xsPath is a directory, else *false*.

Example:

```
if(is_directory(TargetName()))
  throw CTT_Error("not a correct target file");
```

Remark:

In contrast to the corresponding function in boost filesystem *is_directory* returns *false* for an empty string and does not produce an exception. Otherwise the above example would not run in the TETRA working space

10.3.5.8 file_size

Prototype

unsigned int file_size(**const** str& xsPath);

Description

Returns the size of the file in bytes as reported by the operating system. An exception is thrown, if the path does not exist or, if the path is a directory.

Example:

```
if(exists(SourceName()) && file_size(SourceName()) == 0)
   throw CTT_Error("source file is empty");
```

10.3.5.9 find_file

Prototype

bool find_file(**const** str& xsDirectory, **const** str& xsName, str& xsFoundPath, **bool** ci = true);

Description

This function returns *true*, if a file exists in the directory *xsDirectory* with the name *xsName*. *xsFoundPath* then contains the complete path for this file. If the file isn't found, the function returns false. The case of the letters of the name is ignored by default, as it is usual under Windows. With the fourth parameter can be forced, that the case is not ignored.

Example:

```
str sPath;

if(find_file(SourceRoot(), "log.txt", sPath))
{
   ...
```

see also: unit_dependence example

10.3.5.10 load_file

Prototype

bool load_file(str& xs, **const** str& xsFileName)
bool load_file(str& xs, **const** str& xsFileName)

Description

Opens the file with the name *xsFileName* and reads the content into the string *xs*. *true* will be returned, if this process was successful, otherwise *false* will be returned
If the command *load_file* is used, the file is opened in the text mode, i.e. under Windows, that that all CR/LF combinations (carriage return/line feed) are translated into a single LF character. This is not the case, if the file is read in the binary mode with *load_file_binary*.

Example:

```
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;
```

Caution with cyclic Include files. If two files include each other directly or indirectly a serious abnormal end of the TextTransformer can happen. To avoid this danger, the opened files should be logged and a file only should be opened, when it isn't already open.

10.3.5.11 path_separator

path_separator is a string constant for the path separator. The constant contains the value which is used by the operating system to separate lists and files from each other in paths

Windows : "\\"
Unix : "/"

So, e.g.:

```
path = directory + path_separator + filename;
```

results in a correct path for all operating systems, which are covered by the boost filesytem library.

10.3.6 Formatting instructions

Formatting commands are used to determine, how the generated output looks like. (See also: indentation)
The interpreter provides the Boost Format library from Samuel Krempp for formatting arguments according to a format-string(, similar to printf in c).

The syntax of the format string and the right number of the arguments will not be checked before the execution of the formatting command.

Remark: The following explanations are essentially restricted to the "%|spec|" specification. Other possibilities more close to the traditional printf syntax can be found in Krempps original documentation.

A format object is constructed from a format-string, and is then given arguments through repeated calls to operator%. Each of those arguments are then converted to strings, who are in turn combined into one string, according to the format-string.

```
out << format("writing %|1$|,  x=%|2$| : %|3$|-th try")
                          % "toto" % 40.23 % 50;
// outputs: "writing toto,  x=40.230 : 50-th try"
```

10.3.6.1 How it works

1. When you call *format(s)*, where s is the format-string, it constructs an object, which parses the format string and look for all directives in it and prepares internal structures for the next step.

2. Then, either immediately, as in

```
out << format("%|2$| %|1$|") % 36 % 77 )
```

or later on, as in

```
format  fmter("%|2$| %|1$|");
fmter % 36; fmter % 77;
```

you feed variables into the formatter. Those variables are dumped to out, which state is set according to the given formatting options in the format-string -if there are any-, and the format object stores the string results for the last step.

3. Once all arguments have been fed you can dump the format object to out, or get its string value by using the str() member function. The result string stays accessible in the format object until another argument is passed, at which time it is reinitialized.

```
// fmter was previously created and fed arguments, it can print the result :
out << fmter ;

// You can take the string result :
str s  = fmter.str();

// possibly several times :
s = fmter.str( );

// You can also do all steps at once :
out << boost::format("%|2$| %|1$|") % 36 % 77;
```

4. Optionally, after step 3, you can re-use a format object and restart at step2 :
fmter % 18 % 39;
to format new variables with the same format-string, saving the expensive processing involved at step 1.

All in all, the format class translates a format-string (with eventually printf-like directives) into internal operations, and finally returns the result of the formatting, as a string, or directly to out.

10.3.6.2 Examples

Simple output, with reordering:

```
out << format("%1% %2% %3% %2% %1% \n") % "11" % "22" % "333";
```

It prints : "11 22 333 22 11 \n"

Simple output, no reordering:

```
out << format("writing %||,  x=%|| : %||-th step \n") % "toto" % 40.23 %
50;
```

It prints : "writing toto, x=40.23 : 50-th step \n"

More precise formatting, with positional directives:

```
out << format("(x,y) = (%|1$+5|,%|2$+5|) \n") % -23 % 35;
```

It prints : "(x,y) = (-23, +35) \n"

Two ways to express the same thing:

```
out << format("(x,y) = (%|+5|,%|+5|) \n") % -23 % 35;
out << format("(x,y) = (%|1$+5|,%|2$+5|) \n") % -23 % 35;
```

all those print : "(x,y) = (-23, +35) \n"

New formatting feature: 'absolute tabulations', useful inside loops, to insure a field is printed at the same position from one line to the next, even if the widthes of the previous arguments can vary a lot.

```
for(unsigned int i=0; i < names.size(); ++i)
     out << format("%|1$|, %|2$|, %|40t|%|3$|\n") % names[i] % surname[i]
% tel[i];
```

For some std::vector names, surnames, and tel (see sample_new_features.cpp) it prints:

```
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

The format-string contains text in which special directives will be replaced by strings resulting from the formatting of the given arguments.

Remark: The format syntax is leaned on the one used by printf of the c-language. If you are familiar with printf you can look at Samuel Krempps original documentation at www.boost.org, where some differences of the syntax of the Format library to that of printf is discussed. On top of the standard printf format specifications, new features were implemented, like centered alignment.

Format accepts several forms of directives in format-strings :

- Legacy printf format strings : %spec where spec is a printf format specification. This is considered

as obsolete in the TextTransformer.

- %|spec| where spec is a printf format specification. The brackets are introduced, to improve the readability of the format-string, but primarily, to make the type-conversion character superfluous in spec, which are necessary in the first form. (Optionally it still can be used.) e.g. : "%|-5|" will format the next variable with width set to 5, and left-alignment (just like the following printf directives : "%-5g", "%-5f", "%-5s" ..)

- %N%; this simple positional notation requests the formatting of the N-th argument - without any formatting option.

A specification spec has the form: [N$] [flags] [width] [. precision]

Fields inside square brackets are optional. Each of those fields are explained one by one in the following list:

N $ (optional field) specifies that the format specification applies to the N-th argument. (it is called a positional format specification) If this is not present, arguments are taken one by one. (and it is then an error to later supply an argument number)

flags is a sequences of any of those:

Flag	**Meaning**
'-'	left alignment
'='	centered alignment
' '	internal alignment
'+'	show sign even for positive numbers
'#'	show numerical base, and decimal point
'0'	pad with 0's (inserted after sign or base indicator)
' '	if the string does not begin with + or -, insert a space before the converted string

width specifies a minimal width for the string resulting form the conversion. If necessary, the string will be padded with alignment and fill characters specified by the format-string (e.g. flags '0', '-', ..)

precision (preceded by a point), sets the precision

When outputting a floating type number, it sets the maximum number of digits:

after decimal point when in fixed or scientific mode
in total when in default mode

When used for a string argument it takes another meaning: the conversion string is truncated to the precision first chars. (Note that the eventual padding to width is done after truncation.)

%{nt} , where n is a positive number, inserts an absolute tabulation. It means that format will, if needed, fill the string with characters, until the length of the string created so far reaches n characters. (see examples)

 # %{nTX} inserts a tabulation in the same way, but using X as fill character instead of the current 'fill' char (which is space in default state)

10.3.6.4 Methods

unsigned int format::size() const

To get the number of characters in the formatted string, you can use the size() member function :

```
format  formatter("%|+5|");
out << formatter % x;
unsigned int n = formatter.size();
```

str format::str() const

Once all arguments have been fed into a format object, you can get its string value by using the str() member function:

```
format  formatter("%|+5|");
formatter % x;
str s = formatter.str();
```

void format::parse(const str& xs)

Clears the format object and parses a new format string

```
format  formatter("%|+5|");
out << formatter % x;
formatter.parse("%|+10|");
```

10.3.7 Other functions

clock_sec	calculates time
random	generates random numbers
time_stamp	converts the present date and time into a string

10.3.7.1 clock_sec

Prototype

double clock_sec()

Description

Calculates the "CPU time" in seconds, since the program has been started.

clock_sec can be used, to calculate the time, which passed between two events.

clock_sec in principle is: std::clock() / CLK_TCK;

Return value

clock_sec returns the used "CPU time" in seconds, since the program has been started.

If the processor time isn't available or the value cannot be represented, the function returns -1.

10.3.7.2 time_stamp

Prototype

str time_stamp()
str time_stamp(const str& xsFormat)

Description

Converts the present date and time into a string. If the function is called without parameter or with an empty string, then the resulting string contains 26 characters in the following format:

```
Mon Nov 21 11:31:54 1983\n
```

To get another format, the function can be called with a format string. This string is a combination of normal text and formatting characters followed on "%". The possible formatting characters are listed in the following table:

Format character	Meaning	Example
a	Abbreviated weekday name	Sun
A	Full weekday name	Sunday
b	Abbreviated month name	Feb
B	Full month name	February
c	Date and time	Feb 29 14:34:56 1984
d	Day of the month	29
H	Hour of the 24-hour day	14
I	Hour of the 12-hour day	02
j	Day of the year, from 001	60

m	Month of the year, from 01	02
M	Minutes after the hour	34
p	AM/PM indicator, if any	AM
S	Seconds after the minute	56
U	Sunday week of the year, from 00	
w	Day of the week, with 0 for Sunday	0
W	Monday week of the year, from 00	
x	Date	Feb 29 1984
X	Time	14:34:56
y	Year of the century, from 00 (deprecated)	84
Y	Year	1984
Z	Time zone name	PST or PDT

Only those parts that are actually used are noted

Return value

A string with date and time

Examples

```
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;
```

results in:

```
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
```

when writing this help section.

10.3.7.3 random

Prototype

int random(int num)

Description

Number randomizer.

random returns a random number in the area of 0 to (num - 1). Both num and the random number returned are integer values.

Return value

random returns a number between 0 and (num-1).

10.3.8 Parser class methods

Beside of the class methods, which you can define freely, there are some predefined class methods of the parser.

1. Access of the parser state
2. Plugin methods

10.3.8.1 Parser state

At each moment the state of the parsing process is characterized by the actual position in the input text and by the hitherto recognized token and productions. The interpreter can access some of the properties of the actual state. The state as a whole is represented by the variable xState.

Remark to the names ***xState*** and ***State***:

In the course of the development of the TextTransformer ***State*** was also used instead of the name ***xState***. The preceding 'x' shall express that it is a parameter variable. *State* was taken as a class element too. The parser state is only existing as a parameter by now. Therefore the name *xState* is used everywhere now. However, *State* can be used for *xState* as synonymous furthermore. For the interpreter the use of *State* or *xState* doesn't make any difference. See also: xState as parameter of a call of a class method

Single properties of the state can be investigated by the following instructions:

```
unsigned int size() const
unsigned int length(int sub = 0) const
stri 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

bool xState.IsSubCall() const
str ProductionName() const
str BranchName() const

int GetState()
void SetState(int xeState);

Example:

Source text: one two three four
Production: "one" "two" "three" "four"

If "two" was recognized last, is valid:

```
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

Returns the number of sub expressions, which take part at the actual recognition, included the whole recognition (sub expression with the index null). This is the case even if no matches were found

str **str**(int sub) const

Returns what matched, item 0 represents the whole string, item 1 the first sub-expression and so on, defaults to the whole match (sub == 0).

bool **matched**(int sub) const
bool **matched**() const

returns true, if the sub-expression specified by *sub* matched or whether there is a match at all.

str **text**(unsigned int from) const
str **text**(unsigned int from, unsigned int to) const

By the function *text* you get parts of the source text.
If it is invoked with only one parameter, it delivers the text from the position "from" until the end of the token recognized currently. With the second parameter the end of the text section can be determined.
If *from* or *to* is greater than the length of the source text or if *to* is greater than *from,* an empty string is returned. If only *to* is greater as the length of the source text, the string from *from* until the end of the text is returned.

str **copy**() const

returns the string from the end of the last recognized token to the end of the current recognized token. xState.copy() is equivalent to xState.str(-1) + xState.str():

unsigned int **length**(int sub = 0) const

Returns the length of the matched sub expression, defaults to the length of the whole match (sub == 0).

int **LastSym**() const

Returns the internally given number of the last recognized token.

unsigned int **Line**() const

Returns the line number of the last recognized token. (The line count begins with one.)

int **Col**() const

Returns the column number of the last recognized token. (The column count begins with one.)

unsigned int **LastPosition**() const

Returns the position of the last recognized token, that means the number of characters from the beginning of the parsed text to the first character of the recognized token.

unsigned int **Position**() const

returns the position, where last recognized token ends. This position is equal to LastPosition() + length()

unsigned int **NextPosition**() const

returns the position, where the next token begins, that means the number of characters from the beginning of the parsed text to the first character of the expected next token.

If a SKIP token was recognized last, then position and NextPosition are identical.
The spaces at the end of the text covered vy SKIP can be removed with *trim_right_copy*

void **SetPosition**(unsigned int xi);

With *SetPosition* the current position can be set directly as a number of characters from the start of text. The next token is calculated newly with the current scanner in result of this method. E.g. this can be useful if the text contains Meta information about the lengths of its components.

bool **IsSubCall**() const

returns *true*, if actually a production is executed, which was invoked from the interpreter. A temporary parser-state variable *xState* is used here (in contrast to the plugin). Within the productions of the main parser this function returns *false*.

str **ProductionName**() const

returns the name of the actual production.

The return type always is std::string, even, when Unicode parsers are created.

This function is not thread save.

str **BranchName**() const

returns the name of the last branch (alternative, option etc.) in the grammar.

The return type always is std::string, even, when Unicode parsers are created.

This function is not thread save.

---

int **itg**() const
int **itg**(int sub) const
double **dbl**() const
double **dbl**(int sub) const

The functions ***itg*** and ***dbl*** immediately convert the text of the token recognized last into an integer value or a double value. ***itg*** returns a correct integer value for text sections too, which can be interpreted as octal or hexadecimal numbers.

---

str **lp_str**() const
str **lp_copy**() const
str **lp_str**(int sub) const
unsigned int **lp_length**(int sub = 0) const

These methods are concerning the part of text, which was recognized by the last call of a production ("lp" for "last production").
lp_str returns the part of text without the ignored characters at the beginning, while *lp_copy* returns the whole text. The methods *lp_str(int sub)* and *lp_length(int sub)* are formally equivalent to the methods *str(int sub)* and *length(int sub)*. But they can be called only with the indices -1 and 0. If the index is == -1, you get the information about the ignored text of the production.

Example:

```
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 recognizes "a", which then is printed Prod2. Prod2 then recognizes " b c", which is output at the end of Prod1.

---

str **la_str**() const
str **la_copy**() const
str **la_str**(int sub) const
unsigned int **la_length**(int sub = 0) const

These methods are concerning the part of text, which was recognized by the last call of a look-ahead parser ("la" for "look-ahead").

la_str returns the part of text without the ignored characters at the beginning, while *la_copy* returns the whole text. The methods *la_str(int sub)* and *la_length(int sub)* are formally equivalent to the methods *str(int sub)* and *length(int sub)*. But they can be called only with the indices -1 and 0. If the index is == -1, you get the information about the ignored text of the look-ahead.

---

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

This group of functions is analogously to the functions whose names don't start with "next_". They return the corresponding values for the token expected next. The time, when the next token is found out has changed in the course of the development of the TextTransformer, and it might be possible, that modifications could arise again. These functions have to be used therefore only under reservation.

---

int **GetState**()
void **SetState**(int xeState);

There are some integer values which characterize the state of the parse-state.

```
typedef enum { epCleared,
               epExpectingToken,
               epExpectingSKIP,
               epExpectingBreak,
               epExpectingEOF,
               epNoProgress,
               epStopped,
               epExpectationError,
               epUnexpectedError,
               epSkipMatchedNeatless,
               epUnknownError,
               epParsedIncomplete,
               epUnknown
             } EPState;
```

Experienced users can try to manipulate these values in *OnParseError*, to make some error recovery.

10.3.8.1.1 Sub-expressions

Parenthesis "(...)" in regular expressions can be used to mark sections of text, which are recognized by the sub-expressions of a regular expression.

By means of an index parameter for the functions *str* and *length* the text or the length of sub-expressions can be accessed

The whole recognized text is returned by the call of xState.str() or xState.str(0). xState.str(1) returns the section of text, which was recognized by the sub-expression with the index 1.

The index of a sub-expression is the number of its opening parenthesis inside of the whole expression. The counting begins from left to right with the index 1.

In the menu help an item *Regex test* exists, where you can open a dialog, which presents a simple possibility to investigate the sub expressions of a regular expression.

Following indices are defined:

- 2	everything from the end of the match, to the end of the input string
- 1	the ignored characters in front of the actual recognized token
0	the actual recognizes section of text
0 < N < size()	the section of text, which was recognized by the N'th sub expression of the actual recognized token
N < -2 or N >= size()	Represents an out-of range non-existent sub-expression: an empty string

Example:

the expression:

"(ab)*"

may be applied to

"abababab"

Then xState.str(1) would contain the last "ab" of the text.

Sub-expressions can match empty strings. For example a sub-expression can be part of an alternative, which doesn't match text.

10.3.8.2 Plugin methods

The plugin methods are using data, which are valid only for one pass through the parser or are changed dynamically while such a pass:

Paths and names of the input and output files
Redirection of the output
Indentation stack
Text-scope stack
Dynamic scanners

Error handling

The Plugin methods can be used in the interpreter, like normal functions.
Only in the case, that the parser shall be exported as c++ code, there are some points to be taken into account.

If the const option is active, you have to call the methods as methods of the parse state, if however the const option is deactivated both possibitities to call the methods are equivalent. For example:

const is not active:	*ResetOutput*();
or:	*xState.ResetOutput*();
const is active:	*xState.ResetCout*();

The plugin methods are combined to this special group for the creation of multi threaded save code. The plugin methods and data are located in a special plugin class, which is "transported" through the productions by the parse state class. So they can be changed, without any influence on the state of a const parser.

10.3.8.2.1 Source and target

Prototype

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)

Description

By these functions you can get the current source and target directories and files. The use of these functions makes sense at most for the transformation manager, the command line tool and in the generated code.In the TETRA working space these paths are constructed from the current project directory and the name "unnamed.txt", if texts weren't loaded in the source window or saved in the target window before.

str SourceName()

returns the name of the actual source file; the absolute path included.

str TargetName()

returns the name of the actual target file; the absolute path included.

str SourceRoot()

returns the absolute path of the superior source directory

str TargetRoot()

returns the absolute path of the superior source directory.

bool IsLastFile()

indicates, if the actual source file is the last of a sequence of files. This information can be important, if a group of source files is processed to a single target file to finish certain actions. If single source files are transformed to single target files respectively, this function returns *true*.

void SourceName(const str& xsSourceName, bool xbLast)
void TargetName(const str& xsTargetName)
void SourceRoot(const str& xsSourceDir)
void TargetRoot(const str& xsTargetDir)

In the created c++ code the name of the actual source file cannot be evaluated automatically, but the programmer can set it by this function; accordingly the source directory and the target directory. When the name of the source file is set, you can use the second parameter **xbLast** to indicate, whether the file is the last of a sequence of files. Per default xbLast is *true*, because in a 1:1 transformation each source file is the last of the actual transformation.

str str()

If you are not in the TextTransformer IDE and the output wasn't redirected to a file, the output will be written into a buffer of the plugin. The content of the buffer can be retrieved by the str-method of the plugin:

```
xState.GetPlugin().str()
```

When this method is called, the buffer is cleared.

10.3.8.2.2 Start parameters

Parameters which are needed before the start of a transformation can be submitted by the Plugin. Depending on the way of execution parameter strings are set in the code, in the command line, in the transformation manager or in the project options. If necessary, this string can be processed with an sub-parser to take it to pieces..

str **ConfigParam** ( ) const

Parameters for the configuration of an arbitrary transformation can be read with the function *ConfigParam,* if they were set by the user before.
Include directories are a typical example of configuration parameters.

str **ExtraParam** ( ) const

Parameters for the configuration of a certain transformation can be read with the function *ExtraParam* , if they were set by the user before.

10.3.8.2.3 Redirection

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**( )

obsolete:
void **RedirectCout**(const str& xsFileName, bool xbAppend);
void **ResetCout**()

Description

When the command **RedirectOutput** is called, all output will be written into the file with the name, which is specified by the parameter *xsFileName. out* then represents the file *xsFileName*. If a file with this name doesn't exist, it will be created.
Per default an existing file with the specified name will be overwritten. If the second parameter *xbAppend* is *true,* the output will be appended to an existing text.

The command has an effect only within the transformation-manager, the command line program or the exported code. In the working space the output always is written completely into the output window.

You cannot use RedirectOutput, if in the options UTF8 encoding is set. By this option RedirectOutput internally is performed already.

By **RedirectOutput** you can split the target into different files.

By **ResetOutput** the original state is restored. So the output to out is printed into the original file.

It is *neither* necessary to call **ResetOutput** before calling **RedirectOutput** nor at the end of a transformation.

10.3.8.2.4 xerces DOM

XML documents can be produced, processed and written with the DOMDocument class of the Open source project

http://xml.apache.org/xerces-C/

In the TextTransformer the corresponding operations are integrated into the interpreter in such a way, that the DOM-elements, which are wrapped in the *dnode* interface, can be used like node's. Except from the life time of the *dnode*, the only difference compared with the *node* is, that a connection to the *DOMDocument* class must made for the *dnode*. The instantiation of the *DOMDocument* class happens in the CTT_Xerces class and the Plugin can transport a pointer to this class. The connection is made by the single call of the *GetDocumenttElement* function for the root node of a dnode-tree.

dnode **GetDocumentElement**();

To connect dnodes with the DOMDocument of the plugin, code like the following has to be be executed:

```
dnode root = GetDocumentElement();
```

Analogously to the description of the tree construction from *nodes* further *dnodes* now can be added to the root *dnode*

void **WriteDocument**();
void **WriteDocument**(const str& xsFilename);

By means of the *WriteDocument* command the DOM can be issued in the form of XML.

```
WriteDocument();
WriteDocument(const str& xsFilename);
```

xsFilename is the name with a complete path for the file into which the document shall be written. It can optionally be passed to the function *WriteDocument*. Without this parameter the document is written in TargetName. *TargetName* may not be passed as a parameter since otherwise the file would be tried to open a second time. An existing file with the name *xsFilename* is overwritten.

10.3.8.2.5 Indentation stack

A frequent task is to indent some parts of the output. To facilitate this task there is a member in the plugin class, which manages a list (a stack) of values for indentation. One such value denotes the number of characters, e.g. blanks, which shall be sent ahead the real text of a line. A stack of such values is needed, to handle nested indentations.

Example:

Typical cases are indentations to improve the readability of program code:

```
for ( int i = 0; i < 10; i++)
{
  if ( i == 5 )
  {
    func1 ();
    func2 ();
  }
}
```

Here at first the for-loop is indented, then the if-clause and then the function calls are indented further again. When a brace closes, the indentation is set back to the previous value.
If the not indented texts of the lines are stored in a vector (vstr) v, the output shown above can be created by:

```
{{
  PushIndent(2);  // indent the whole text by two characters

  for( int i == 0; i < v.size(); i++)
  {
    if( v[ I ] == "}";  // if a closing brace follows, set the indentation back
      PopIndent( 2 );

    out << indent << v[ i ];

    if( v[ I ] == "{";  // increased indentation after an opening brace
      IncrIndent( 2 );
  }
}}
```

The central instruction of this example is:

```
out << indent << v[ i ];
```

indent is a class element, which contains the values for the indentations, but also can be used - as shown here - for direct formatting of the output. For this it is passed to a stream, that means, by being inserted in the chain of output elements connected by "<<". Normally, this insertion will take place at the first position. Otherwise gaps would arise instead of indents.

indent can only be used inside of a <<-chain.

str **IndentStr**() const

By this function you get a string, which only consists of spaces. The length of the string is determined by the value on top of the indentation-stack.

Methods, which determine the indentation value, are:

void **SetIndenter**(char xc)

Per default blanks are used for the indentation. With the function *SetIndenter* another character can be set, e.g. the tabulator: '\t'.

void **PushIndent**(int xi)

By this instruction the new value *xi* for indentation is pushed on the stack.

void **IncrIndent**(int xi)

By this instruction also a new value for indentation is pushed on the stack. The new value is the sum of the value on top of the stack and *xi*.

void **PopIndent**()

The value for indentation on top of the stack is removed.

void **ClearIndents**()

All values are removed from the indentation stack.

10.3.8.2.6 Text-scope stack

By means of the scope stack you can record, which part of the text just is processed. Parts of text can be the introduction, a heading, a subparagraph etc. or the declaration part or the corresponding definition part of a class in a programming language.
Dynamic scanners, which are presented on the next page, can "remember" tokens, which belong to a certain scope.

Methods of the text-scope stack

void **PushScope**(const str& xs)

By this method the name of a part of text is put on the stack.

void **PopScope**()

By *PopScope* the uppermost value of the stack is removed.

void **ClearScopes**()

Removes all values from the stack

str **ScopeStr**() const

ScopeStr returns the name of the part of text, which is stored on the top of the stack.

10.3.8.2.7 Dynamic scanner

Because of dynamic scanners the TextTransformer is capable of learning.
Parts of text (literals), which are recognized by a general regular expression, can be assigned to a placeholder token. If the same part of text appears again in the input, the dynamic scanner can recognize it.

Since the next token always is already recognized, a newly added dynamic token can be recognized only if the next token was consumed. If necessary, xState.*SetPosition(xState.Position())* can be invoked after *AddToken*. So, a new recognition is forced at the current position.

Since TextTransformer 1.3.4 *AddToken* can be executed as a transitional action. The token is then already available before the determination of the next token.

Independently of the project settings placeholder tokens always are case sensitive.

bool **AddToken**(const str& xsText,
const str& xsDynTokenName)

AddToken adds the string *xsText* as an alternative to the placeholder token *xsDynTokenName*, which must have been defined in advance on the token-page. If then the parser tests for an occurrence of the token *xsDynTokenName* in the text, the test will match, if *xsText* or another string added by *AddToken* is found.
The function returns *true* at success. If the placeholder-token isn't defined, *false* is returned.

bool **AddToken**(const str& xsText,
const str& xsDynTokenName,
const str& xsScope)

The relatively complex *AddToken* method with three parameters has been designed especially to parse programming languages.
If this method is called, *xsText* will be added as an alternative like in the case of a call of *AddToken* with only two parameters. But this alternative is recognized in the text only, if the actual text scope is *xsScope* or a scope subordinated to *xsScope, that means* added after *xsScope*.
If for example the declaration of a class is parsed, the names of the class variables can be assigned to the class - that means to the name of the class. In the definition part the name of the class can be set again and the dynamic scanner will recognize the variable names. Outside of the class scope the same names can have a different meaning.

Example: " Eval; Eval; Eval "

```
ID
{{
AddToken(xState.str(), "USER_FUNCTION", "FUNCTION_SCOPE");
PushScope("FUNCTION_SCOPE");
PushScope("BLOCK_SCOPE");
}}
// the next token already is recognized, therefore Eval can be recognized the
next but one as USER_FUNCTION.
```

```
";"

USER_FUNCTION
// Eval will be recognized as a USER_FUNCTION, because the actual scope
BLOCK_SCOPE is subordinated to FUNCTION_SCOPE.

{{
PopScope();
PopScope();
}}


";"

USER_FUNCTION
// Eval will not be recognized, because actually there isn't any scope set;
especially FUNCTION_SCOPE and none of its subordinated scopes aren't set.
```

Example:

In some grammars of other parser-generators the alternatives of a syntax rule are listed in such a manner, that the name of the production and the definition symbol "::=" is put in front of each of them. E.g.:

```
Rule1 ::= A B
Rule1 ::= C Rule1

Rule2 ::= D
Rule2 ::= Rule1 E
```

If such rules shall be imported into the TextTransformer, then dynamical tokens can be used, to combine the alternatives into a single rule:

```
{{ 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)

With this function all dynamic tokens which were defined for the text area xsScope are deleted. If

xsScope is an empty string, all tokens for all areas are removed.

If tokens were assigned to certain text areas, *ClearTokens* should be invoked with an empty string before finishing a program to avoid memory leaks.

10.3.8.2.8 Error handling

void **UseExcept**(bool xbUseExcept)

By this command you can determine, whether the parsing in case of a bug shall be interrupted by an exception (xbUseExcept == true) or, whether the error message is stored intermediately in a container of the plugin class and the abort is done by the fact, that for the next token the symbol number null is given back.

bool **GetUseExcept**() const

returns, whether the parsing in case of a bug will be interrupted by an exception or, whether the abort is managed by returning the symbol number null for the next token.

bool **HasError**() const

HasError returns *true*, if a parsing bug has occurred, which hasn't triggered an exception, because *UseExcept* was set to false. If *HasError* is *true*, you have the chance to execute some action immediately after the error occurred, before the parsing aborts.
After the execution of a production that was invoked directly from the semantic code every error, which might have happened hereby is deleted, so that *HasError* returns *false*.

void **GenError**(const str& xs)

By the call of GenError an error is created, which leads to an abort of the parsing. A string that contains the error message is passed to the function. This report is displayed in the log window after the abort of parsing.
If *UseExcept* is not set the parsing will be finished regularly, otherwise the abort is done by an exception similar to the throw command. The exception, which is created by *GenError* however contains additional information about the location of the error.
.
void **AddMessage**(const str& xs)
void **AddWarning**(const str& xs)
void **AddError**(const str& xs)

The plugin class now has per default a container (vector), by which messages, warnings and errors can be assembled For this there are the three methods: *AddMessage*, *AddWarning* and *AddError* with a text string as parameter. These strings are shown in the log window after the parsing has finished or they appear in the result window of the transformation manager, if the project was executed there.

In the case of errors, some also special events occur which can be used for the treatment of errors..

In the exported c++ code you get two iterators by the methods *MsgBegin* and *MsgEnd* by which you can access the messages, which are derived from *CTT_Message*. By *HasMessage* can be proved whether there is a message in the container and by the method *GetMsgType* of *CTT_Message* you get one of the enumerated types: *eMessage*, *eExit*, *eWarning*, *eError*, *eParseError* or *eSemError. An example of the use of the iterators can be seen at the end of the main-frame.*.

10.3.9 Calling a production

In the introduction the similarity of productions and functions was shown. The TextTransformer uses this similarity by not using productions only within the central parser framework but by also allowing the direct use of productions as functions. Productions can be used

1. as independent sub parsers
2. for a look-ahead

10.3.9.1 Sub parser

A production can be called directly from the interpreter code. It then is not part of the real grammar of the parser in which this interpreter code is embedded. The called production is rather a start rule for a separate parser and a new input text is passed to it explicitly. The **string parameter** makes the difference to the look-ahead call of a production.

For example a sub parser could extract some parameters from a part of text, which was recognized by a look-ahead parser, before the main parser continues:

```
IF( Production() ) // look-ahead parser
  {{ Parameters( xState.la_str() ); // sub parser }}
  Production // main parser
END
```

Other applications of sub parsers are the treatment of include-files or external data could be fed into the program by a sub parser.
If e.g. the production with the name *Include* shall be called, then this could be done as follows.

```
{{
  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");
}}
```

Possible parameters of the *Include* production are appended to the string for the new source text.

To determine the number of lines to be parsed already before the beginning of parsing, the sub-parser CountLines can be used:

```
CountLines(int& xi) ::=
(
  SKIP EOL
  {{ xi++; }}
)+
```

CountLines would have to be called in an action which is executed before the first token of the main parser was recognized:

```
{{
int iLines = 0;
CountLines(xState.str(-2), iLines);
}}
```

10.3.9.2 Look-ahead

Productions can be invoked in the interpreter to test whether the input text matches the production at the current position. If you use the look-ahead in combination with the IF symbol or by WHILE, you can parse texts, which aren't LL(1) conform or which are difficult to describe with a LL(1) grammar.
By this look-ahead the current position isn't changed and the semantic code of the production isn't executed. Parameters therefore aren't necessary. The TextTransformer just recognizes a look-ahead call by the missing parameter, opposed to the call of a sub parser, which needs at least a string parameter.
The text is tested either until a token is expected which doesn't exist in the text or until the last symbol of the production is recognized. So a **bool value will be returned** which is *true* if the text matches the production and *false*, if it doesn't match. Under no (logical) circumstances exceptions are thrown.
A look-ahead can depend on other look-ahead productions which are tested inside of the first. The different levels of the look-ahead are represented as a number both in a separate field of the tool bar and as a preceding number in the stack window.

The part of text, which was recognized by the look-ahead can be accessed by the la_str() method of the parser state class. For example, you can pass this string to a sub parser.

A detailed example is the Java parser.

The result of the look-ahead depends on whether the look-ahead production is part of the system where it is called or not.

The production *Ident* is used as a look-ahead production in the same parser system, where it is

used as a normal production. The input "int" yields the expected result: "int found", if the option for testing all literals is activated.

```
IF(!Ident())
 "int" {{cout << "int found"; }}
ELSE
 Ident  {{cout << "error"; }}
END

Ident ::= IDENT
IDENT ::= \w+
```

If, however, an external production is used for the look-ahead:

```
isIdent ::= IDENT

IF(!isIdent())
 "int" {{cout << "int found"; }}
ELSE
 IDENT  {{cout << "error"; }}
END
```

the input "int" yields the error: "IDENT" expected. Since *isIdent* is a start production which isn't used in the main parser, it doesn't know the tokens of the main parser either. So "int" is nterpreted as *IDENT* and !isIdent() is wrong. The ELSE alternative is chosen. However, "int" was recognized in the main system as int what isn't accepted by *IDENT* then.

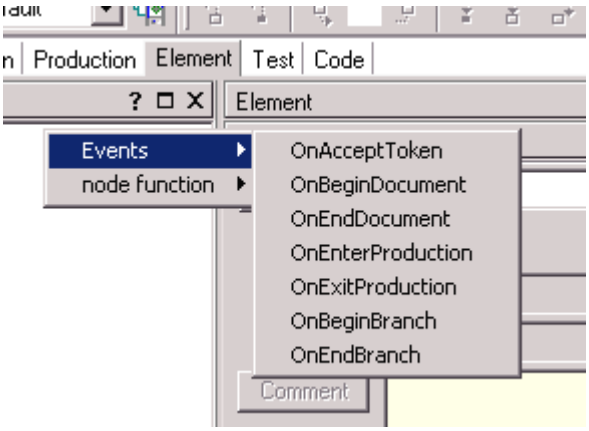
In other cases it is to use an external production for looking ahead is of advantage. So in the next example: ***SKIP*** will find the character at the end of a sentence even if the sentence starts with "What" or "How".

```
isQuestion ::=
SKIP
(
    "?"
  | ( "." | "!" )
     EXIT
)

Sentence ::=
IF(isQuestion())
  Question
ELSE
  NonQuestion
END

Question ::=
(
    "What" {{ out << "You should better know than I!"; }}
  | "How"  {{ out << "I don't know how!"; }}
)
| {{ out << "I don't understand your question!"; }}
IDENT+ "?"
```

```
NonQuestion ::=
IDENT+ ( "." | "!" )
{{
out << "That's interesting!";
}}
```

10.3.10 Events

There is a number of functions which are always called automatically when a certain event appears. In these functions nothing happens as long as a corresponding event handling wasn't programmed explicitly. The first step, to do this is, to insert such a function by the pop-up menu of the list on the element page:



In contrast to the functions of the TetraComponents of the same name no explicit parameters are passed in these functions. Like in all TETRA functions, there is, however, the implicit xState parameter, by which you can access all properties of the parser-state

OnEnterProduction

The event *OnEnterProduction* occurs, when the parser branchs into a production

OnExitProduction

The event *OnExitProduction* occurs, when the parser leaves a production.

OnAcceptToken

The event *OnAcceptToken* occurs, if a token recognized by the scanner is found and accepted in the grammar. This happens in the debugger at the moment where a terminal node is left.

OnBeginBranch

The event *OnBeginBranch* occurs, when the parser enters an option or a repeat.

OnEndBranch

The event *OnEndBranch* occurs, when the parser leaves the option or a repeat.

OnBeginDocument

The event *OnBeginDocument* occurs, when the parser begins with a new input.

OnEndDocument

The event *OnEndDocument* occurs, when the parser finishs parsing a file or a string.

OnParseError

The event *OnParseError* occurs before the parsing is aborted with an error message. Sometimes the abort can be prevented, if the error can be recovered here. Otherwise you have the chance in *OnParseError* to create some additional report about the circumstances of the error.

10.4 Test scripts

Test scripts are constructed similar to the scripts of productions or token. But test scripts cannot have a return type and instead of an optional parameter list here exists an optional input text. The syntax of a test scripts is identical with the syntax of a production.

The mask for the definition of a test script has following fields:

Name:	unique name
Comment:	arbitrary comment
Input:	input text
Text:	script
Expected output:	Expected output
Test output:	Real output of the executed test

Name and text are needed. If one of these fields is empty, the script will not be accepted and you can't write a comment.

10.4.1 Name

Each test script must have a name.
A name can be constructed of the alphanumeric characters and the underscore, but the latter may not be at in first place of the name.

Examples: test1, int_test, CALC

Each new name must differ from all other names of tests by at least one character, but it has not to be different from the token and production names.

10.4.2 Group

This is an optional field for the name of a group of tests.
All test scripts with the same group name are combined to one group. The sub rules of all tests of a group are compiled together. If Test group or Test all is executed. This is faster than compiling all sub rules for each test again.

In rare cases the result of an isolated test may be different to the test inside of a group. The follow set of a SKIP symbol may change, because inside of a different test additional token may follow.

10.4.3 Comment

A comment to a test can be shown in the yellowish field. Temporary this field is also used to show error messages.

To change the comment, use the button. A dialog will be opened, where you can write the new text.

10.4.4 Input

In the field for the input optional a text can be written, which shall be transformed by the test rule. The test processes the input in the same manner, as an input text on the main page of TETRA would be processed by an according start rule.

10.4.5 Code

The syntax of a test scripts is identical with the syntax of a production. Because a test normally is used to test a production, a test script often consists in the declaration of variables, which are passed as parameters to the tested production.

10.4.6 Expected output

In the field *Expected output* the text has to be written, which the execution of the test should produce. After execution, this text is compared with the real output and in case of a deviation an error message will be created.

10.4.7 Test output

The field for the test output is a read only field. While creating a new test, this field remains empty. After the execution of a test the generated output is written into this field and compared with the expected output. In case of a deviation an error message will be created.

10.4.8 Error expected

Sometimes the parsing of a text shall fail. If a failure is expected, the check box in the tool bar should be activated.

If an error occurs while executing the test of an expected failure, the error message is shown in the output window, but the error will not be listed in the error list.

Part XI

11 Algorithms

It is helpful to know the algorithms, which the TextTransformer uses for text analysis, to develop efficient programs and for better understanding possible errors and conflicts. As explained already in the introduction this analysis is done in two steps:

For the **lexical analysis** an algorithm to extract the next token by a scanner is needed

For the **syntactical analysis** an algorithm for the choice of the next branch in the system of rules (grammar) by the parser is needed

11.1 Scanner algorithm

A scanner tries to find the next token that matches the actual text. In contrast to other parser generators the TextTransformer not only uses one scanner, but can use a variety of little scanners, which are specifically testing only a sub set of all possible token. These sub sets consists of the first set of the actual node and the SKIP-alternatives. However the principle of the scanner remains the same: for each token of a set is tested, if it matches or not. If there are several matching tokens, an algorithm exists to decide between the candidates. In detail:

1. Testing the first set

The TextTransformer tries to find a match of the actual text and one of the tokens of the first set of the actual node. If there is exactly one matching token, at this point the analysis is finished.

2. Preference of the longest match

If there is more than one match at the same text position, the token with the longest match will be preferred; that means the token, which covers the greatest number of characters.

3. Preference of literal tokens

If there still several token are possible, literal tokens are preferred.

4. Preference of the longest match of the first sub match

If two candidates aren't literals, the token with the longer first sub match is chosen.

5. Preference of the longest match of the next sub match

If there is still an ambiguity the length of the matches of the further sub expressions will be consulted as criterion.

6. SKIP alternatives

If there is no matching token in the first set, SKIP alternatives will be tested. Is there exactly one match the lexical analysis at this point is finished.

7. next text position

If there is more than one candidate, to which can be skipped, the token is chosen, which matches at the nearest position in the text.

8. longest match (analog point 2-5)

If there are several token of the SKIP set, which all match at the same position of the text, the decision is made according to the longest match analogous to point 2-5.

It is important, that no definition of one of alternative tokens includes the definition of a different. The TextTransformer cannot give any warning in this case.

Example:

```
IDENT = \w+
NUMBER = \d+
```

Because "\w" among others includes the set of numbers, the definition of IDENT also includes the definition of NUMBER. If there is an alternative

```
IDENT | NUMBER
```

and a number is at the input position, it isn't clear, which alternative will match (->remark). The alternative of IDENT and NUMBER must not be formulated explicitly. A problem results in every case, where conflicting tokens are hidden in a common first set.
Generally it is recommended to define token as specifically as possible. A definition of

```
IDENT = [[:alpha:]|_]\w*
```

would avoid the problem mentioned above. IDENT cannot begin with a digit and NUMBER must begin with a digit.

Remark: Indeed IDENT will match. and if IDENT were defined as "[[:alpha:]|_|\d]+" NUMBER would match. But this is determined by the internal implementation of the Regex-Library and cannot be specified by rules.

11.2 Parser algorithm

The task of a parser is the syntactical analysis of text. The parser uses a rule system (**grammar**), which defines the valid possibilities of token sequences. At every new step of parsing is tested, whether the actual evaluated token is a token, which may follow or not. If not, the parsing failed; if yes, the parsing will be continue through the alternatives of the grammar, which is determined by the

token.
The decision about the validity of the next token is made by means of precomputed token sets: the first sets of the possible following nodes, including the SKIP alternatives. Here the function of the parser is crossed over with the scanners, because these token sets are exactly the sets of the single scanners, which were the theme in the previous section. In the project options the set of tokens, which will be tested can be limited to the set of allowed tokens. This results into an increased velocity of parsing, especially, if the lexicon is big.

The parsing algorithm is as follows:

1. Testing the first set of the startrule

At the beginning of the text analysis no token is recognized. Because of that, a local (interface) scanner compares the beginning of the text with the set of tokens, by which the text can start, according to the grammar.

2. Going to the expected terminal symbol

If a match is found, the parser steps to the node with a first set that contains the evaluated token. This may be repeated through a sequence of nullable nodes, until the node is reached, which represents the token itself.

3. Testing the followers inside and outside of the production

Each terminal node has a scanner, which evaluates the next token. But now there are two cases:
a) inside of the production, which contains the actual terminal symbol there always are other terminal symbols situated, which must be passed before it is possible to leave the actual production. In this case the next token will be evaluated and continued as described at point 2,
b) inside of the production, which contains the actual terminal symbol nothing follows or only nullable structures are following. In this case, the set of possible followers of the terminal symbol depends on the followed of the production. But a production can be called at different places of the grammar, and at every place may be followed by different token sets. In this case the TextTransformer will test at first only the incomplete set of those token, which can follow inside of the actual production. Further point 3 will be executed.

Note:
The TextTransformer is an interpreted parser generator. It is a parser, which parses production scripts to create a new parser. The created parser in turn parses input text to transform it. Because the parser of the TextTransformer is created by itself, the algorithms of parsing scripts and of parsing input text are the same.

11.3 Token sets

The decision on the next branching in the grammar can be made dependent in special cases of a look-ahead in the text or of the semantic predicates. In most cases however, which will discussed now, the grammatical alternative is chosen whose first symbol is recognized in the text next. Besides the preference rules already explained this can depend on the set of the tokens which are tested: tokens which aren't tested cannot cause any conflicts either. Therefore there is the

possibility in the options of the project and in the local options, only to test expected tokens. However, it can just be desired to recognize conflicts early, too. So e.g. reserved words of a programming language may not be used for variables as names:

```
double int;  // error
```

The option is to test expected tokens only in this case has to be deactivated.
However, the token sets have to be discussed at the use of productions outside of the main parser.

Token sets of inclusions, sub-parsers and in a look-ahead

It's conspicuous for the case of comments that conflicts with the tokens of the main parser are unwanted.

```
CppComment ::= "/*" ( SKIP | STRING )* "*/"
// ! this definition isn't appropriate for nested comments

/* int iCount : Zähler */
```

If the keyword *int* were recognized here, the comment couldn't be parsed.
Inclusions can therefore form a new production system in the TextTransformer which is independent of the token set of the main parser.
Exactly the opposite applies to look-ahead productions: it is mostly desired here, that the same tokens as in the main parser are recognized.

The following rules are allowing a flexible adaptation of the token sets to the respective purposes:

1., Any production which isn't called by another production, i.e. every start production, is a base for an independent production **system** with a token set of its own. This is the union of all tokens occurring in the system. Look-ahead productions used in the system aren't regarded as called here, so they are not part of the system automatically). The start rule of the main parser is the first system.

2., If a production is used in several systems, then the token sets of these systems are united.

Example:

```
Prod1  ::= IF( Prod2() ) Prod2 ELSE Prod3 END
Prod2  ::= "a" "b"
Prod3  ::= IF( !Prod4() ) "c" "d" ELSE  ID+ END
Prod4  ::= SKIP "h"
```

The rules *Prod2* and *Prod3* are reached from the start rule Prod1. The token set of this system is: "a", "b", "c", "d", ID.

Prod2 is used as a look-ahead. Since this production, however, also is part of the system of the start rule, the set of tokens tested by global scanners in *Prod2* is identical with that one of the main system: "a", "b", "c", "d", ID.

The look-ahead with *Prod4*, however, is independent of the main system. *Prod4* therefore forms a system of its own whose set of tokens only consists of "h".

Now let's extend *Prod4* to:

```
Prod4  ::= SKIP "h" Prod2?
```

Prod2 then would be a part of the two previous systems. In this case the token sets of the two systems are united in accordance with point 2 above: "a", "b", "c", "d", "h", ID.

This expansion of *Prod4* can have to consequence that e.g. "a" isn't skipped by *SKIP* in the text "a h" any more.

Part XII

12 Grammar tests

A TextTransformer program must suffice to the condition of the so-called LL(1) analysis, which will be explained in this chapter. If one of these conditions is not fulfilled, the TextTransformer creates an error message or a warning. In the case of an error message, the error must be eliminated, otherwise the program can neither be executed nor code can be generated. In the case of a warning, the program can be executed and code can be created, but there are conflicts, which the TextTransformer resolves by an automatic choice. Generally the TextTransformer chooses the possibility, which the syntax analysis meets at first. In rare cases this resolution contradicts to the intentions of the program author:

In detail a TETRA grammar must fulfill the following conditions.

1. The grammar must be complete
2. Rules must be reachable
3. The grammar may not be circular
4. Rules must be derivable to terminals
5. The grammar must be LL(1)

12.1 Completeness

Each non-terminal (=production) must be defined by exactly one rule and there may not be a terminal, which is not defined

The TextTransformer guarantees the first part of the condition automatically, because it is not possible to insert two productions with the same name.

For the second part however the TextTransformer will execute a special test. It may happen, that a non-existing symbol name was used in the definition of a production. Already the syntax highlighting helps to avoid such an error: this name will not be shown in boldface printing. Not later than the TextTransformer parses the script an error message will be produced:

```
Unknown symbol: "xxx".
.
```

12.2 Reachable rules

Normally LL(1) parser like TETRA must fulfill the condition, that each production must be reachable from the start rule.

In TETRA however this principle is weakened. TETRA's family concept allows the existence of productions in the repository, which are not all subordinated to a common start rule. Nevertheless this condition is guaranteed during the execution of a TETRA program or for code production,

because the TextTransformer cuts out exactly the set of rules, which are derivable from the actual chosen start rule.

12.3 Derivable rules

Each non-terminal symbol must be derivable to terminal symbols. The following production is an example for a rule, which hurts this principle:

```
X ::= "(" X ")"
```

If you parse this rule, following error message will occur:

"X": can't derive to terminals

12.4 Non-circularity

No non-terminal may be derived from itself, neither directly nor indirectly. The following production is an example for a rule, which hurts this principle:

```
Circular1 ::= Circular2 | "T"
Circular2 ::= Circular1 | "T"
```

If you parse this rule, following error message will occur:

Circular derivation: "Circular1" . "Circular2"
Circular derivation: "Circular2" . "Circular1"

Remark:
Following rules however are yielding the error message, no being derivable to terminals.

Circular1 ::= Circular2
Circular2 ::= Circular1

The test of derivability is done before the circularity is found and the second test will not be performed.

12.5 LL(1)-Test

Recursive descent parsing requires that the grammar of the parsed language satisfy the LL(1) property. This means that at any point in the grammar the parser must be able to decide on the bases of a single look-ahead symbol which of several possible alternatives have to be selected. For example, the following production is not LL(1):

```
ident ":=" Expression
| ident ( "(" ExpressionList ")" )?
```

Both alternatives start with the symbol "ident", and the parser cannot distinguish between them if it comes across a statement, and finds an "ident" as the next input symbol. However, the production can easily be transformed into

```
ident
(
  ":=" Expression
  |  ( "(" ExpressionList ")"  )?
 )
```

where all alternatives start with distinct symbols. There are LL(1) conflicts that are not as easy to detect as in the above example. For a programmer, it can be hard to find them if he has no tool to check the grammar. The result would be a parser that in some situations selects a wrong alternative. The TextTransformer checks if the grammar satisfies the LL(1) property and gives appropriate error messages that show how to correct any violations.

If LL(1) conflict can't be resolved by restructuring the grammar, you can use the IF...ELSE...END or WHILE...END structures.

12.6 Warnings

The following messages are warnings. They may indicate an error but they may also describe desired effects. The generated compiler parts may still be valid. If an LL(1) error is reported for a construct X, one must be aware that the generated parser will choose the first of several possible alternatives for X.

Following warnings can appear:

1. "X" is nullable
2. LL(1) error: "X" is the start of several alternatives
3. LL(1) error: "X" is the start and successor of nullable structures

.

12.7 Nullability

The warning

"X" is nullable

appears, if the production "X" matches an empty text. For example "X" may be defined by:

```
X ::= "la" *
```

This rule will recognize a text like "la la la", but also the empty text "" matches, because the

'*'-repeat matches null repeats. In contrast, the production

```
X ::= "la" +
```

is not nullable.

This warning can be a hint, that a nullable alternative was created inadvertently.

Productions, which only consist of semantic actions, in principle, are nullable too. However a warning will not be created in this case, because such a rule cannot lead to a wrong recognition, but can be uses as helping function.

The creation of this warning can be suppressed by an according checkbox in the project options.

12.8 Start of several alternatives

The warning

LL(1) error: "X" is the start of several alternatives

appears, if alternatives are binning with the same terminal symbol. For example in the following rule:

```
Greeting ::= "good" "morning" | "hello" | "good" "evening"
```

the first and the last greeting are beginning with the word "good". If TETRA finds this word in the input, the first alternative would be chosen, even if the word "tag" were following. The rule has to be rewritten as follows:

```
Greeting ::= "good" ("morning" | "evening" ) | "hello"
```

12.9 Start and successor of nullable structures

The warning

LL(1) Warning: "X" is the start and successor of nullable structures

appears, if a nullable structure begins with the terminal symbol "X" and the same terminal symbol can follow this structure. Nullable structures are (...)? and (...)*. An example is the following rule for recognition of a (German) number with possible fractional digits:

```
Number ::= ( Digits "," )? Digits
//LL(1) Warning: Digits is the start and successor of nullable structures
```

If the parser meets some digits, the nullable structure will be tested. If no comma follows the program will be finished with an error message. The rule should be written reversed:

```
Number ::= Digits ( "," Digits )?
```

Because it depends on the chosen start rule which productions will be parsed, it also can depend on the start rule, if a token is start and successor of nullable structures or not.

An example of this sentence can be obtained by a little extension of the last example. So it's conceivable that the number production is part of a number pair production in which two numbers shall be separated by commas. E.g.:

```
Pair  ::= Number "," Digits
```

This is a serious conflict.

The creation of this warning can be suppressed by an according checkbox in the project options.

Another example is known in the literature as ***dangling else***. The following two rules are hurting the LL(1)-condition:

```
Statement ::= IfStatement | ...

IfStatement ::= "if" Expression "then" Statement
                    ( "else" Statement )?
```

The instruction:

if a then if b then c else d

would be interpreted by TETRA as follows:

```
if a then
{
   if b then
      c
      else
      d
}
```

that means, TETRA would arrange the *else*-branch to the *if b*. In a logical point of view the following arrangement would be possible too:

```
if a then
{
   if b then
      c
}
else
d
```

12.10 SKIP node with SKIP neighbors

The error message

"X" is a SKIP node with SKIP neighbors

appears, if in one alternative more than one SKIP node is contained.

Until now, in TETRA has no algorithm to treat such cases.

Example:

```
Rule1 ::=
  (
  "TETRA"
  | SKIP
  )*

Rule2 ::=
  (
  "Texttransformer"
  | SKIP
  )*

Rule3 ::= Rule1 | Rule2
```

A simple reformulation resolves the problem here:

```
  Rule3  ::=
  (
  "TETRA"
  |"Texttransformer
  | SKIP
  )*
```

The situation is more difficult, if the neighbored SKIP nodes belong to different kinds of structures.

12.11 Different SKIP followers

Enter topic text here.

12.12 Different ANY followers

Enter topic text here.

12.13 Left recursion

Another possible error of a grammar is the left recursion. The following rule is the simplest example of a left recursion:

a = a | B

To test this rule leads to an infinite regress. To test **a**, requires to test **a** before **b** and so on. Left recursion is not permitted in TETRA; but there is no automatic test for left recursion.

Fortunately left recursion is avoidable. The left recursive rule
:

a = a C | B

can be rewritten. Clearly *a* has to begin with *B*. On *B C* can follow and on *B C* another *C* can follow. So the formal result is:

a = B C *

Applied to the first example, C is an empty symbol, with the result:

a = B

Remark:
Productions of the well-known parser generator **Yacc** frequently are left recursive. For Yacc this is no problem, because it doesn't uses a top down analysis like TETRA, but a bottom up parser. (Because of this Yacc can't manage right recursion.) To translate a Yacc grammar into a TETRA grammar, the rule above helps.

12.14 Circular look-ahead

A look-ahead cannot be executed, if it is circular

E.g. the following look-ahead would be obviously circular:

```
expression  ::=
IF( expression() )
...
```

In the look-ahead *expression* is tested again and in this test once more etc..

However, the circularity also can be hidden like in the following productions:

```
expression  ::=
IF( factor() )
...

factor  ::=
IF( expression() )
...
```

TETRA tries to detect such circularities and creates according error messages.
As an additional security procedure the stack is limited for the look-ahead productions.

Part XIII

13 Code generation

In the professional version of TextTransformer you can export source code for a c++ class of the created parser. TextTransformer produces all code, which is required for an independent executable program. The generated code can as well be used as a part of a larger application .

The use of the generated code has three advantages compared to the use of a project with the TextTransformer:

- the program made with the code can be used independently from the TextTransformer
- the program is faster
- the semantic actions aren't restricted to the subset of the C++ instructions, which can be interpreted by the TextTransformer

In the introduction already was explained, that a production could be considered as a specification for the creation of o function. The generation of code now is the application of this specification, i.e. for each production an according function will be built.

The c++ code for the semantic actions is written into the generated code, according to the project option, simply by copying or by reconstruction. The code for the parser will be written at indicated positions in a code frame.

To compile a parser generated by the TextTransformer you need the library of regular expressions from Dr. Maddock and the some supporting C++ code:

13.1 Code frames

The code for the parser will be written at indicated positions in code frames.
The name of the generated parser class is derived from the name of the start rule.
There is

1. a frame for the header
2. a frame for the implementation
3. a frame for the call of the parser

The given standard frames can specifically be adapted to the project.

In addition a file with the name *jamfile.txt* is generated, which helps to create a "jamfile" for the use with with boost bjam.

13.1.1 Name of the parser class

The name of the production, which is chosen as start rule, is the basis for the name of the generated parser class.

An upper 'C' will precede this name and the word "Parser" will be appended.

Example: If the name of the start rule is "Pascal", the name of the parser class will be:

"CPascalParser":

Remark: the 'C' is for "class". This corresponds to the convention in Visual C++.

13.1.2 Header frame

The unmodified frame for the header is as follows:

```
//-------------------------------------------------------------
//   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
```

The arrow "-->" and the following key word indicate the positions, where the TextTransformer will write the code.

-->ParserHeaderSentinel will be replaced by an expression, which is constructed from the name of the start rule of the parser. For example, if this name is "Pascal" the sentinel will look as follows:

```
#ifndef PascalparserH
#define PascalparserH
…
#endif // PascalparserH
```

-->**ScannerEnum** denotes the position, where some enumeration types are defined. The values of the enumerations are indices for the differen mini-scanner.

-->ParserClassName is a dummy for the name of the parser class.

The second occurrence of **-->ParserClassName** denotes the constructor of the parser class. Into the following parenthesis additional parameters can be inserted.

StartRuleDeclaration denotes the public function to call the parser. The name of this function is constructed from the name of the start rule. For example:

"void Pascal(cts xtBegin, cts xtEnd);"

cts is a typedef for std::string::const_iterator

InitProcDeclaration will be replaced by:

```
void Init();
```

This is the declaration of a procedure, by which the member variables are initialized.

InterfaceDeclarations is a dummy for the sequence of declarations of functions, by which the parser can be called (beneath the start rule). In the local options *create interface* must be activated, to stimulate the creation of these functions. The declarations have the same form as the declaration for the start rule.

ParserRuleDeclarations denotes the position, where the list of declarations for the productions and sub-classes to scan the input are inserted. So for each production, there will be a function of the same name. Especially the start rule will be declared here:

"void Pascal(tetra::sps& xState);"

sps is a typedef for CTT_ParserState, a class, which represents the state (especially the positions) of the parsing process, and which is passed and actualized from rule to rule.

13.1.3 Implementation frame

The unmodified frame for the implementation of a parser class is as follows:

```
//-------------------------------------------------------------
//   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 becomes to *Pascalparser.h*, remaining at the example of the previous page.

-->**TokenList** will be replaced by a list of the names and definitions of the tokens. This list helps to understand the following code.

-->**ParserClassName** has the same meaning as explained above. Inside the parenthesis behind the second occurrence of this dummy, you can write declarations of parameters according to those in the header.

-->**MemberInitialization**, is used to for the initialization of member variables, as e.g. functions tables.

-->**StartRuleHeading** corresponds to **ParserCallDeclaration** of the header frame. This is the heading of the public function to call the parser.

-->**StartRule** is a placeholder for the function block of the function, by which the parser is called. You can include it by a try catch block, if desired.

-->**InitProcImplementation** is the place, where the procedure will be inserted, by which the class variables are initialized.

-->**InterfaceImplementations** is the dummy of the sequence of interface methods, created because of the activation in the local options.

-->**ParserRules** finally is the position, where the implementations of the functions will be written, which are corresponding to the productions.

13.1.4 main-file frame

A main function is produced per default for a console application with the following main file frame.

```
//-------------------------------------------------------------
//    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;
}
```

The program expects two parameters: one for the source path and one for the target path.

There are five new placeholders in the frame which are not in the other frames

XercesInclude
XercesUsingNamespace
XercesInit
LoadFile
OstreamType

The first three placeholders are deleted if dnode isn't used in the project.

-->**LoadFile** is substituated by:

```
load_file_binary( sTest, pSourceName)
```

for binary open mode of the source file or by

```
load_file( sTest, pSourceName)
```

-->**Ostream** is substituded either by *ofstream* or by *wofstream*, depending on the activation of the selected wide-char option and the flag *ios::binary* is set according to the open mode for the target file.

If dnode's are used, the plugin-type CTT_ParseStateDomPlugin has to be set.

-->**XercesInclude** includes tt_xerces.h

->**XercesUsingNamespace** inserts the following line:

```
using namespace xercesc;
```

-->**XercesInit** creates an instance of *CTT_Xerces* and passes a pointer to it to the 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 Project specific frame

For each project you can create specific frames, if you want. It will be saved at the address specified in the project options.
In this frame for example class members or methods can be defined or additional include directives can be written. You can use the dummy explained above. For example a global pointer to an instance of the parser could be defined as follows:

extern -->ParserClassName* _-->ParserClassName;

It is important, that the character immediately following "**-->ParserClassName**" is not alphanumeric, because otherwise the dummy would be ignored.

13.1.6 jamfile

When the c++ code is generated, in addition a file with the name *jamfile.txt* is generated, which helps to create a "jamfile" for the use with with the boost build system.

http://www.boost.org/doc/tools/build/index.html

Followingly, an example of such a file is represented, which was produced for a project with the start rule *translation_unit* and which uses dnode's.

```
# 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
;
```

If CTT_ParseStateDomPlugin is not set as plugin-type the lines concerning xerces don't appear in *jamfile.txt*. The boost filesystem library isn't always required either. However, this isn't automatically checked.

13.2 Supporting code

To compile a parser generated by the TextTransformer you need the code of some additional classes, which are used in the projects: Most classes are situated in header files alone ("header only"), i.e. their source code doesn't have to be included into the projects of C++ compilers explicitly. The structure of the tetra directory shows, which files must be included: those of the *tetra/source* directory namely.

The complete list of the classes is:

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

An implementation of the helping functions

stod
stoi
dtos
itos
etc.

can be found in tt_lib.h/cpp

13.2.1 Code directory

The supporting code for TextTransformers parsers is in the directory *tetra*. The structure of this directory is orientated at the model of the *boost* libraries and reflects, which files must be included in C++ compiler projects.

```
tetra
  |
   ---  config
  |
   ---  source
  |       |
  |          ---  xercesdom
  |
   ---  xercesdom
```

Only the files of the subdirectory source must be included and if dnode's are used, also the directory *source/xercesdom*. All files in the root directory *tetra* and in the directory *tetra/xercesdom* are header files.

13.2.2 CTT_Parser

template <class char_type> **class** CTT_Parser

CTT_Parser a simple base class for the parser generated by the TextTransformer. The template parameter either can be *char* or *wchar_t*.
When required you can derive your own class from the generated parser and overwrite some of its virtual methods, for example the parser events.

13.2.2.1 Methods

The virtual methods of the class *CTT_Parser* are partly overwritten in generated parsers or can be overwritten by the developer in a class of his own.

```
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 represents the actual state of a parser. The class contains a boost::match_results<iterator>-member, with the information about the last match and iterators marking positions in the input text.
The first template parameter either can be *char* or *wchar_t*. The second template parameter is either CTT_ParseStatePlugin or a type derived from it.

13.2.4 CTT_Scanner

template <class char_type> **class** CTT_Scanner

Instances of the class CTT_Scanner are organizing the extraction of the next tokens at the decision points of the parser. The template parameter either can be *char* or *wchar_t*.

CTT_Scanner distributes his task over local scanners:
CTT_IgnoreScanner: removes ignorable characters
CTT_LiteralScanner: tests on literals
CTT_DynamicScanner: tests on dynamic literals
CTT_RegexScanner: tests on regular expressions
CTT_SkipScanner: tests on the next occurrence of a token in the text.

13.2.5 CTT_Tst, CTT_TstNode

template <class char_type> **class** CTT_Tst

CTT_Tst is the implementation of a ternary search tree for literal tokens.
CTT_TstNode is a node in the ternary search tree.
The template parameter either can be *char* or *wchar_t*.

13.2.6 CTT_Match

template <class char_type > **class** CTT_Match
: **public** CTT_Token<char_type>

CTT_Match is a class, to remember the result of a token test. As opposed to the base class CTT_Token it contains a *match_results* element from the boost regex library.

The template parameter either can be *char* or *wchar_t*.

13.2.7 CTT_Token

template <**class** char_type > **class** CTT_Token

The class CTT_Token is used, to remember the result of a token test. As opposed to the derived class CTT_Match it doesn't contain a *match_results* element and therefore is used to store the recognized tokens in the token buffer efficiently.

The template parameter either can be *char* or *wchar_t*.

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>

When a text is parsed at least the token recognized last and the next token are buffered. The buffering happens with these classes.

CTT_BufferLL1ex contains three CTT_Match elements: two for the tokens just mentioned and perhaps a third token to which is jumped by SKIP.

CTT_BufferAll contains a Stack from CTT_Token to buffer the look-ahed tokens.

The template parameter either can be *char* or *wchar_t*.

13.2.9 CTT_Guard

template <class char_type, class plugin_type> **class** CTT_Guard

At the beginning of every production the constructor of an instance of the class CTT_Guard takes care that a stack is updated which consists of the scanners - more exactly: pointers to scanners - which has to be tested at the end of the call of a production.When leaving the function, the destructor of CTT_Guard actualizes the stack again, by removing the last added scanner. The destructor is used since other instructions wouldn't be executed any more after a return instruction.

The first template parameter either can be *char* or *wchar_t*. The second template parameter is either CTT_ParseStatePlugin or a type derived from it.

Macros:

The code produced by the *TextTransformer* shall be portable, i.e. it shall work on different systems and with different c++ compilers. Because *Microsoft Visual Express C++* behaves differently than other compilers, it is required to include the complete code of a production into a try-catch block - by means of two macros - so that the call of the destructor is actually carried out at the desired time.

```
#define  ENTER_GUARD(number,  production)  \
   GUARD  Guard  =  ProductionBegin(xState,  xiScannerIndex,  xiSkipScannerIndex,  xeLS,  number,  produ
  try {
```

(ENTER_CONST_GUARD and ENTER_LA_GUARD are corresponding macros for const parsers and lookaheads.)

```
#define  EXIT_GUARD(number,  returnvalue)  \
       } \
```

```
catch (...) { \
    if(xiScannerIndex > -2) \
       throw; \
} \
 Guard.StayAlive(); \
 return returnvalue;
```

An "optimization" takes care in *Visual express C++* that the *CTT_Guard* variable is destroyed again immediately after its creation. To keep the variable at life up to the moment, where the production is left, the *Microsoft* compiler must be led to believe that the variable would be needed again. Therefore the otherwise useless *StayAlive* function of the *CTT_Guard* class is called after the catch-block. This function only will be passed, when the production doesn't return a value. Otherwise the return is already carried out within the try-catch block. If an exception is actually catched, it is thrown again in any case. The condition:

```
if(xiScannerIndex > -2)
```

is always true, as result of the code generator. However, for the compiler the *StayAlive* call seems to be reachable.

Without the last line of the macro the code wouldn't compile.

```
return returnvalue;
```

If the production doesn't return a value, *returnvalue* remains empty. If the production, however, returns a value, a default value must be known for the return type although the line is actually never executed. This value can therefore be arbitrary as long as it matches the corresponding return type. For the interpreter code the return value is generated automatically. But if a return type is defined in code only for the export a default value must be given. This can be done in the field for the return type by appending a slash and the value. E.g.::

```
{_ CProduktion* _}/NULL
```

13.2.10 CTT_Mstrstr

template <class char_type> **class** CTT_Mstrstr

CTT_Mstrstr has the same interfaces as mstrstr of the interpreter. Inside of the generated code mstrstr is defined by:

typedef CTT_Mstrstr<char> mstrstr;

CTT_Mstrstr is derived from std::map<Key, T, Compare, Allocator > and so also has the interfaces of a 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 is derived from std::map<Key, T, Compare, Allocator >. By this class, assignments of pointers to parser class member functions to names are stored.

13.2.12 CTT_Node

template <class char_type> class CTT_Node

CTT_Node is a class written especially for the TextTransformer, which represents the type node in the exported c++-code and which is used in the code of the TextTransformer itself.
The template parameter either can be *char* or *wchar_t*.

13.2.13 CTT_DomNode

class CTT_DomNode;

dnode's in the interpreter are defined as typedef of CTT_DomNode in the exported code.
CTT_DomNode capsules a xercesc DOMElement with a xercesc DOMText child. So you can use CTT_DomNode in the same manner as CTT_Node, The name of the DOMElement represents the label of the node and the value of the DOMText is the value of the node. If you write a CTT_DomNode into a XML file, you get:

<label>value</label>

While the memory is internally managed for a CTT_Node by reference count of all nodes of a tree, a CTT_DomNode is managed by xerces.

If you use dnodes, you have to chose CTT_ParseStateDomPlugin, and the Xerces library has to be linked to the produced code.

Before a parser is called in the generated c++ code, the default label has to be set. *CTT_DomNode* has a static method for this purpose.

```
dnode::SetDefaultLabel(L"default_label");
```

13.2.14 CTT_ParseStatePluginAbs

template <class char_type >
CTT_ParseStatePluginAbs<char_type>

Abstract basis class for CTT_ParseStatePlugin and for CTT_ParseStateDomPluginAbs.

13.2.15 CTT_ParseStatePlugin

template <class char_type> class CTT_ParseStatePlugin

CTT_ParseStatePlugin is the class for the plugin. From this class a user defined class can be derived, which contains methods and data, which are valid during exactly one pass of the parser.
The template parameter either can be *char* or *wchar_t*.
The plugin type has to be set in the project options.

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

This class has the same data and functions as CTT_ParseStatePlugin. In addition it contains a pointer to a CTT_Xerces class, which capsules a xerces DOMDocument.
The plugin type has to be set in the project options. If you use dnodes, you have to chose CTT_ParseStateDomPlugin, and the Xerces library has to be linked to the produced code.

13.2.18 CTT_RedirectOutput

template <class char_type > **class** CTT_RedirectOutput

This class manages the redirection of the output into files.
The template parameter either can be *char* or *wchar_t*.

13.2.19 CTT_Indent

template <class char_type > class CTT_Indent

A simple indentation stack. The template parameter either can be *char* or *wchar_t*.

13.2.20 CTT_Xerces

class CTT_Xerces

This class capsules a xerces DOMDocument and contains some functions to use it.
A pointer to an instance of this class is contained in CTT_ParseStateDomPlugin. So the document can be created in the parser and used afterwards..

```
  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 Error handling

Functions for the treatment of errors are listed at the plugin methods. Further there are four events, which force the parser to abort and there is a treatment routine for each of these events, which normally throws a corresponding exception type. These methods are virtual and can be overwritten.

```
void   OnErrorExpected(state_type&  xParseState,
                       int xiSym,
                       const char* xpProduction,
                       const char* xpBranch) const;
```

The token with the number *xiSym* was expected, however, wasn't it found in the production with the name *xpProduction* at the branch with the name *xpBranch*.

```
void   OnErrorUnexpected(state_type&  xParseState,
                         const char* xpProduction,
```

```
                                const char* xpBranch) const;
```

Another token was expected, however, wasn't it found in the production with the name *xpProduction* at the branch with the name *xpBranch*.

```
void    OnErrorIncomplete(state_type&  xParseState,
                          const char* xpProduction,
                          const char* xpBranch) const;
```

The input couldn't be parsed completely by the production with the name *xpProduction*.

```
void    OnErrorStandstill(state_type&  xParseState,
                          const char* xpProduction,
                          const char* xpBranch) const;
```

The parser is in an endless loop in the production with the name xpProduction at the branch with the name xpBranch. (This bug cannot occur at the moment, since only tokens covering at least one character are allowed.)

CTT_Message is the basic class for all messages, warnings, errors and exceptions. It contains a message string and details on the current status of the parser

```
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)
```

From CTT_Message directly or indirectly derived are:

CTT_Exit
CTT_Warning
CTT_Error
CTT_ParseError
CTT_ErrorExpected
CTT_ErrorUnexpected
CTT_ErrorStandstill
CTT_ErrorIncomplete
CTT_SemError
CTT_NodeError

13.4 Compiler compatibility

The generated C++ code is essentially portable. It was tested successfully with the following compilers:

Windows

Borland CBuilder 6
Borland CodeGear C++ Builder 2009
Visual C++ 2008 Express

Linux

gcc 4.2

Perhaps smaller customizations of the code frames and of the supporting code could be required with other compilers and other systems.

The c++ code generated by the TextTransformer uses the **library of regular expressions** of Dr. Maddock at

http://www.boost.org/

The library conforms to the stl and was tested nearly with all actual operating systems and different compilers. Details can be found at

http://www.boost.org/development/tests/trunk/developer/summary.html

Xerces-C++ is portable too. A list of the supported operations systems and compilers is at:

http://xml.apache.org/xerces-c/

For TETRA Xerces-C++ was made with Borlands CBuilder 6. To avoid a conflict with dll also delivered by Borland, the suggested name "XercesLib" was changed in "**TTXercesLib**".

13.5 License

result in severe civil and criminal penalties and will be
prosecuted to the maximum extent possible under the law.

RESTRICTIONS

THIS SOURCE CODE AND ALL RESULTING INTERMEDIATE FILES
(OBJ, DLL, BPL, ETC.) ARE CONFIDENTIAL AND PROPRIETARY TRADE
SECRETS OF DR. DETLEF MEYER-ELTZ. THE REGISTERED DEVELOPER IS
LICENSED TO DISTRIBUTE THE TEXTTRANSFORMER LIBRARY AS PART
OF AN EXECUTABLE PROGRAM ONLY.

THE SOURCE CODE CONTAINED WITHIN THIS FILE AND ALL RELATED
FILES OR ANY PORTION OF ITS CONTENTS SHALL AT NO TIME BE
COPIED, TRANSFERRED, SOLD, DISTRIBUTED, OR OTHERWISE MADE
AVAILABLE TO OTHER INDIVIDUALS WITHOUT WRITTEN CONSENT
AND PERMISSION FROM DR. DETLEF MEYER-ELTZ.

CONSULT THE END USER LICENSE AGREEMENT FOR INFORMATION ON
ADDITIONAL RESTRICTIONS.

Part XIV

14 TetraComponents

The TetraComponents allow to use TextTransformer projects in Delphi and CBuilder programs. They are executed (interpreted) with the freely available tetra_engine.dll. The components encapsulate this dll.

Among other things some parse-events can be handled by the components. Thereby numbers are passed for tokens and productions (in OnAcceptToken and OnEnterProduction).

These enumeriated values are written by default into the frame "enums_pas.frm":

```
unit -->NameSpace;

interface

type

  EToken = (
  -->TokenEnums
  );

  EProduction = (
  -->ProductionEnums
  );


const

   Tokens := TStringList.Create;
-->TokenNames
  );

   Productions := TStringList.Create;
-->ProductionNames
  );

implementation

end.
```

The arrow "-->" and the following key word indicate the positions, where the TextTransformer inserts special code for a project. The frame can be edited.

Part XV

15 Messages

In this chapter some warnings and error messages are explained, which can occur, when parsing or executing a project.
If such messages occur, they are listed in the error box. The complete text is show by a click on the according item. The message appears in the central message window. If the item is clicked once and the **F1** button is pressed, the according text of this help is shown.

15.1 Unknown symbol: "xxx"

The test of completeness failed.

15.2 "X": can't derive to terminals

Not all non-terminal symbols are derivable to terminal symbols.

15.3 Circular derivation: "X" . "Y"

There is a circle in the grammar.

15.4 "X" is nullable

A nullable structure was found:

15.5 LL(1) Error: "X" is the start of several alternatives

The grammar is not LL(1)-conform.
.

15.6 LL(1) Warning: "X" is the start and successor of nullable structures

The LL(1)-test has found a possible conflict.

15.7 "X" is a SKIP node with SKIP neighbors

The grammar test has found a SKIP node with unpredictable followers.

15.8 Nullable structure in a repetition or option

The meaning of a **"(...)+"** repeat or a **"(...)*"** repeat or an **"(...)?"** onption isn't clear if it contains a nullable structure.

Examplee:

```
( "a"? )+ "b"
```

The structure "("a"?)+" could be as interpreted as nullable. An occurrence of *b* without previous "a" would be accepted. But you could argue too that the loop has to be passed once at least.

```
( "a"* {{iCount++;}} )* "b"
```

How often should *iCount* be incremented for several "a" in the input? Thar's not clear!

15.9 "X" is used circulary in a look-ahead

This message appears if a look-ahead cannot be executed because it is circular.

15.10 Inclusion not found

A not existing inclusion production is set in the project options or in the local options of a production.

15.11 Conflict with an inclusion

The token, which follows in the grammar is also the beginner of an inclusion production.

15.12 No matching next token found

None of the actual allowed tokens can match the actual text.

15.13 The rest of the text consists of ignored chars

This message points out, that some ignorable characters - according to the project options - remain at the end of the input text, without being copied to the output text. This message is only a hint and doesn't signalize an error.

15.14 SKIP token matches at actual position

This message is shown in the log window, if a token was recognized at the actual text position, which only was expected at a position behind some text, which should be skipped.

Example:

```
SKIP "end"
```

applied on the text

```
"end ..."
```

the SKIP symbol only is accepted, if there is text containing of not ignorable characters between the actual position and the token:

```
"... end"
```

15.15 "SKIP ANY" is not possible

An explicitly defined set of tokens is required for SKIP.

15.16 Matching but not accepted token

This message is shown in the log window, if a token was recognized in the text that in accordance with the grammar isn't permitted.

Example:

```
Input:    Text2Html.ttp
Rule:     SKIP? EOL
Message:  Matching but not acceptet token: ID ( 7 ), \w+
```

"Text2Html" has been recognized as an identifier *ID*, though the whole "Text2Html.ttp" should have been skipped.

As a way out either the option of global scanners for regular expressions can be disabled or the rule can be redrafted:

```
Rule:     ( SKIP | ID )* EOL
```

15.17 Matching token not in first set

This message is shown in the log window, if a token was not chosen, because it doesn't belong to the actual first set.

Example

15.18 Matching look-ahead xxx cannot start with yyy

This message appears if a look-ahead was successful but the token, by which it started, is not contained int the first set of the IF-branch. This can be the case, when a token has been recognized in the main parser, which is not contained in the first set of the IF-branch, but the same text is nevertheless recognized from other tokens in the look-ahead.

15.19 Unexpected symbol in ...

A syntax error inside of a production occurred. A part of text cannot be interpreted. Either a not defined word was used or a permitted keyword can be misspelled. Often wrong brackets or quotation marks are the cause. An expected parameter possibly is missing.

15.20 Parenthesis are needed

You get the warning: *Parenthesis are needed*, if the first semantic action contains a variable declaration for the whole production, for productions of the kind

```
{-...-}
A | B ...
```

Frequently the first semantic action contains declarations of variables, which shall be used in all alternatives. But then, you have to put the alternatives into parenthesis. E.g.:

```
{- str s; -}
(
  "a" | "b"
)
{-return s;-}
```

Without parenthesis

```
{- str s; -}
  "a"
| "b"
{-return s;-}
```

you get the error message:

Unknown identifier: s

for {-return s;-}, because there are implicit parenthesis like

```
(
  {- str s; -}
    "a"
)
|
(
  "b"
  {-return s;-}
)
```

15.21 Unexpected method (also might be ...

The message:

Unexpected method (also might be member-function, which returns a value)

appears, if an instruction begins with an identifier followed by a dot and then followed by a symbol, which does not denote a method. E.g.

```
s.unknown
```

This message can result from a wrong chaining of methods. E.g. *clear* is, a method without return value. Therefore you can't append a second method call on *clear*.

```
s.clear().clear();
```

This message also appears, if you tried to call a member-function, which returns a value; e.g. for the string s:

```
s.length();
```

The following would be correct:

```
int i = s.length();
```

In the syntax of c++, this is correct. But the instruction makes no sense, because the returned value is not used. So here an error message appears.

15.22 "X" expected

According to the grammar a token of the type "X" should have followed the last recognized token in the text. If for example the rule for a salutation is:

```
"Hello" Name "!"
```

and the text is: "Hello !",

this error message appears, because after "Hello" a name is expected.

15.23 Incomplete parse

says, that the parsing of the input has been canceled before reaching the end of the text.

The message "Incomplete parse" refers to the grammar of the parser. This was not processed completely before the input text ended. That means, that at the end of the text further tokens had been expected.

15.24 Missing closing quotation mark

A literal token is immediately defined inside of a production, but the closing quotation mark is either forgotten or it is not in the same line as the opening quotation mark.

15.25 Literal tokens may not be empty

Empty strings are no token. Particularly tokens directly defined inside of a production must consist in one character at least.

15.26 Continuation with c++ code expected

This error message appears, if the bracket of a semantic action was not closed. The production parser recognizes this by the beginning of a new action. For example:

```
{{
str s;
// missing closing bracket
( expression[s] )*
{{
return s;
}}
```

15.27 The type of the function xxx doesn't match the function table

All functions of a function table must have the same return type and the same number and types of parameters.

15.28 No default function is defined for function-table

Each function table must contain a default function. The default function is inserted into the table with an empty string as key. The default function is called, if a node is visited by the *visit* method and the value of the label of the node is not contained as a key in the function table.

15.29 In a const parser you have to call the according method of State

If the const option is set in the project options, plugin methods cannot be called directly. Instead, you have to call them as methods of *xState*.

Example:

instead of

```
str s = AddToken("Print". "USERFUNCTION");
```

you have to write

```
str s = xState.AddToken("Print". "USERFUNCTION");
```

15.30 Sub-expressions (> 0) are not stored in the la-buffer

You cannot access the sub-expressions marked by brackets in the semantic actions if the look-ahead tokens are buffered

15.31 A production cannot be used as an inclusion

A production cannot be used as an inclusion, if it is used by another production (unless a recursive call inside of the inclusion).

15.32 Inclusion with paramters

A production which starts an inclusion may not have parameters. A grammar can be "interrupted" by an inclusion in an arbitrary position. Therefore there isn't any place at which a parameter could be submitted. To exchange information between the main parser and an inclusion, class variables can be used.

If source code for const parser is produced, it is advisable to use variables of the plugin for the information transport.

15.33 Inclusions don't work with a la-buffer

You cannot use inclusions if the look-ahead tokens are buffered

15.34 State parameter is required

This message only appears, if the check box xState parameter for member functions is checked in the project options on the register page warnings/errors.
Then *xState* must be the first parameter in a call of a user defined class method. Only if this is the case, the code of the interpreter also can be exported as c++-code.

15.35 Empty alternative

The complete error message is:

Empty alternatives must be combined with a semantic action

A detailed explanation of this point is given on the page about alternatives inside of productions.

15.36 Error while parsing parameters

A message like:

xxx: Error while parsing parameters "unexpected symbol" in: type_specifier_Alt0

appears, if the text in the parameter field of the rule "xxx" was not parsed correctly. This may be caused by a writing mistake of the parameter type or the interpreter may not know the type. Eventually the parameter was not intended for the interpreter, but for the export. (See: Parameter and {{...}}).

15.37 Mismatch between declaration and use of parameters

This error message (or warning, see below) appears either, if a production or token is called with a wrong number of parameters or if the types of the parameters are not convertible.

For example a parameter declaration of the production XXX may be:

```
Parameter: int xs
```

and the production is called by:

```
{{str s = "Hello";}}

XXX[s] // wrong parameter type
```

A variable of the type string cannot be converted to an "int"-value. Also to leave out the parameter results in the same message.

```
XXX // missing parameter
```

As a warning this message appears if the production is used within a look-ahead production. In look-ahead productions no semantic actions are carried out and not therefore no parameter is used either. In complex projects, however, it can happen that the same production is used also with semantic actions, what may lead to an error only at execution of the transformation program.

To be sure, that no error can happen, you can use dummy parameters.

15.38 Wrong number of (interpretable) arguments

A script, for which a special number of parameters were defined, is called with another number of arguments.
For example for the production *Name* two parameters could be defined:

Parameters: *str* sFirstName, *str* sLastname

If this is called by:

```
Name["John"]
```

the error message appears.

The reason of the message also can be, that an argument passed to the production is not interpretable. If in the project options is set, that not parenthetic text only is copied to the exported code, the following code results in the error:

```
Parameter: {= star str, str sLastname =}

Call: Name["John", "Smith"]
```

15.39 Not const method

The message:

Not const method *xxx* called for const object *XXX*

appears, if the object *XXX* has been declared as const, but the method *xxx* of the object, would modify it.

Example:

In a function with the

```
Parameter:  const mstrstr& xm
```

the cursor of *xm* shall be set to the next value:

```
xm.gotoNext();
```

But the cursor and its position "belong" to mstrstr. So the map would be modified by this action. The parameter has to be declared not const.

```
Parameter:  mstrstr& xm
```

15.40 Maximum stack size of "x"exceeded

This message appears, if the value for the internal stack, which is set in the project options, was exceeded.

Remark: The internal stack here is greater than the shown stack, because it in addition contains the branches to sub rules.

15.41 Error on parsing parameters of the parser call

This message results from an error, which only can occur, if the start rule of the parser needs parameters. In this case, the names of the variables must be extracted from the parameter declaration, notwithstanding if the declaration is interpretable or not.

Example:

The name of a start rule may be "Startrule" and its parameter declaration may be

```
int xi
```

So the following code will be created:

```
void CStartruleParser::Params(ctsr xtBegin, cts xtEnd, int xi)
{
  sps xState(xtBegin, xtEnd);
  Startrule(xState, xi);
}
```

The name "xi" was extracted from the declaration "int xi". If this operation had caused an error the message "Error on parsing parameters of the parser call" had been appeared.

15.42 There is at least one path on which no string value is returned

For a production or a token a return type *xxx* may be declared, but there is a possibility, to walk through the code of the script, without returning a value.
For example a token script could be used to return different strings depending of a recognized number:

```
str sDefault;
switch(xState.itg())
{
  case 1:
  return "eins";
  case 2:
  return "zwei";
  default:
  sDefault = xState.str();
}
```

In the default case, no value will be returned.

15.43 Recognized, but not accepted token

If the execution of a TETRA program was broken by an error, there may be a hint to a matching but not recognized token in the log window.
Presumably an **error** occurred inside of the interpreter before the parser could accept the token. Or a **return-statement** was put in front of the ascertainment of the next token inadvertently.

Example:

```
ID
{{ return true; }}
NUMBER
```

A number NUMBER is expected after the first identifier ID here. However, NUMBER cannot be recognized since the production is exited before by *return*.

Another possibility is, that a global scanner for regular expressions was used, although two tokens of this scanner can **conflict**.

If, for example, a little table consists of the columns version and price of a product and these tokens are defined as:

```
Version = \d\.\d\d
Price = \d\d?\.\d\d
```

that means a price can have one or two digits in front of the point and a version only one. The content of the table can be parsed by:

```
Table = ( Version Price )*
```

If a global scanner is used, it is not clear, whether a number with one digit in front of the dot will be recognized by *Price* or by *Version*. If the number is in the first column, nevertheless *Price* might recognize it. In this case the parsing is stopped and the error message occurs, that Price matched, but was not accepted.
By using local scanners, this is no problem. A number in the first column will be always only tested by *Version* and a number in the second column always only by *Price*.

15.44 BREAK outside of a loop

The **BREAK** symbol only can be used inside of a loop (...)* or (...)+ and in the same production as the loop itself. Using it at another position causes the error message, when the production is parsed.

15.45 Standstill

Despite of parsing, the position in the input wasn't changed.

A token, which matches the text, must match at least one character. A token like

```
Token A:: = a*

Production ::= A
```

would match at any position of the text, because the character 'a' will be there at least null times. A production like (A)* would cause an endless loop. An optional occurrence of a concatenation of 'a'-characters should be written as:

```
Token A ::= a+

Production ::= ( A )?
```

15.46 Standstill in look ahead

There is the danger of an infinite recourse in a look-ahead if at the beginning of a production just this production is used for the look-ahead in an IF condition. Example:

```
ab       =  IF( ab ) ...
```

Before a look-ahead is started therefore will be checked whether the parser already is in a look-ahead. If this is the case and if there hasn't been progress in the text in the look-ahead yet, the parser is stopped and the error message appears:

Standstill in look-ahead

15.47 Unknown identifier : xxx

The message "Unknown identifier : xxx" is caused by an error of the interpreter. A variable was used, which had not been declared before or the scope, where it was declared is left.
This message also can appear as a consequence of another previous error.

EXAMPLE

The assignment

```
s = "Hallo";
```

can't be used, without preceding declaration of s:

```
str s;
s = "Hallo";
```

or combined to one instruction:

```
str s = "Hallo";
```

For the interpreter invisible export code

The error message appears, if the declaration wrongly was set into the brackets for export code "{_" and "_}:

```
{_str sComment;_}
```

while the variable is used in the interpreter code:

```
(
 Comment[sComment]
)+
```

If **interpretable** is activated in the project options Parameter and "{{...}}"
sComment will be seen by the interpreter, but isn't declared.

Hidden Scopes

A variable exists as long as its scope exists. A scope is the whole text of a production/token or the section included into the braces '{' and '}' or defined indirectly by alternatives.
For example in the text of the production:

```
if( xi == 1 )
{
  str s = "one";
  out << s;
}
s = "two" // error
```

the string s doesn't exist any longer after the closing '}'. So nothing can be assigned to it there.

Attention: Alternatives define scopes, which are not visible directly.

Example

The following production causes the error message "Unknown identifier: s", although the string s seems to be correctly declared:

```
{{str s; }}
 "a"
| "b"
  print[s]
```

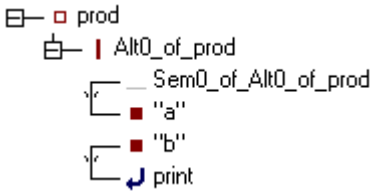
But by the alternatives hidden scopes are introduced:

```
{{str s; }}
 "a"
```

and

```
  "b"
  print[s]
```

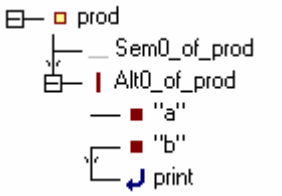
have their owns scopes respectively. This becomes clear in the syntax tree of the production:



The semantic action, which declares the string, is executed immediately before the recognition of "a" and is not preceding the whole alternative. To achieve this, the alternatives must be enclosed into parenthesis:

```
{{str s; }}
(
 "a"
| "b"
  print[s]
)
```

Now the syntax tree looks as:



The declaration now precedes all alternatives and the variable s can be accessed in the whole production.

15.48 It's not possible to convert "xxx" to "yyy"

This message is caused by an interpreter error. The attempt to assign a value to a variable failed, because the value is not convertible to the type of the variable.

Example

```
int i = "Hallo";
```

A String cannot be converted to an integer.

15.49 No return type declared

In the interpreter code a value is returned by *return*, but the field of the return type of the script is empty.

15.50 "X" cannot be applied on "Y"

The interpreter creates this error, if an operation "X" is tried to be applied to a type "X", which is not defined for this type.
For example:

```
str s;
s++;
```

15.51 break or continue instruction at invalid position

A *break*- or a *continue* instruction can be used only inside of the loop of a for-, while-or do-while -statement. *break* also is used in switch-statements. At other positions the use of *break* and *continue* is invalid.

This error only is recognized, while the program is running.

15.52 forbidden transitional action

Only such transitional actions are permitted which leave the state of the parser unchanged before and after use of the production for a look-ahead. There is at least one function in chain of the transitional action for which this isn't ensured.

15.53 Error output programmed from the user

Messages beginning with:

Error output programmed from the user:

stems from the interpreter instruction:

```
throw CTT_Error("message");
```

15.54 Cannot add branch

This message appears in the log window and finishes the execution of a program, if a node-object, which is already contained in a tree, is tried to be added again.

15.55 Token error

There is an error in the definition of a regular expression on the token page.

15.56 Matches empty string

Token may not match an empty text -> standstill

15.57 Token is defined as string and as token with an action

The warning:

"xxx" is defined as string and as token with an action

happens, if a token is defined as well inside of a production as a string, as, associated with an action, on the token page. Principally you can use both inside of one project, if they are not direct alternatives. But often is hardly to discover, whether this is the case or not. So it is recommended to use only one definition. Otherwise unintentional (non-) execution of the action could occur.
If there is no action associated with a token, this message will not be shown.

15.58 boost::regex error

By this message in the log window the execution of a TextTransformer program is stopped, if an error occurs in the class for regular expressions of the boost library.
Presumably the cause is, that the expression is too complex. Remedy is the Friedl scheme, or, if necessary you can split literal parts of the expression and shift the whole expression into a production.

15.59 System overlap

Productions can be used in a fourfold way in the TextTransformer. If a certain production is used in multiple way, then the set of the tokens that is tested inside of it, is determined by use parsed first.

15.60 Token action or member function cannot be exported

A token action or member function cannot be exported, if it consists of parts of interpretable code interrupted by a part of code, which only is exportable.

Example:

```
{= ... =}   // interpretable and exportable code
{_ ... _}   // code only for the export
{= ... =}   // interpretable and exportable code
```

15.61 Only code for initializations is allowd here!

Simple code for the initialization of a variable may be written only into the In the text field of the element page. Other code will produce the error message: Only code for initializations is allowed here.

15.62 Parameters and local variables may not be used in a look-ahead production!

This error message appears, if parsing with a production is production is used directly or indirectly - as well in the main parser as for a look-ahead and the production is controlled by a parameter or by semantic code which accesses a variable declared locally. This is not permitted, because semantic code can produce side effects like a doubled output in look-ahead productions: So all semantic code is ignored, as far as it doesn't immediately belong to the condition of an IF or WHILE structure.

Example:

```
IF( Prod() )
  ...
```

The production Prod is used in an IF structure as a look-ahead production. Prod contains a WHILE structure, iwhich depends on the locally declared variable b.

```
Prod ::=
{{
bool b = true;
}}

WHILE ( b )
... {{b = false;}}
END
```

The use of class variables is tolerated in such cases, 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. Also such use of class variables can hurt const conditions in the generated code.

15.63 Encoding cannot be written into the output window of the IDE

This message appears in the output window if the encoding of a XML document cannot be represented there.

15.64 An invalid or illegal XML character is specified

This message is shown in the log window, if a character was used for the definition of a *dnode*, which isn't allowed there.

Example:

```
nLine.add(",",   xState.str()); // => runtime error
```

15.65 TextTransformer not registered

Index operations are permitted only in the registered version of the TextTransformer.
Index operations are operations, by which you access single elements of a structure. Examples are the access of elements of the container class mstrstr or of the sub-expressions of the recognized token. What exactly may be done with the free version can be seen in this help.

15.66 Internal error: ...

Error messages, which are introduced by the words "Internal error", hopefully should not appear to you. They are created, if internal conditions of the program are not fulfilled. They point out an error of the TextTransformer itself.
If such an error occurs, please tell it to me, including the circumstances, which led to the error. Please use the Feedback form. By this you help to improve the software.

15.67 No help

Unfortunately, it might happen, that after pressing the F1 button no help window appears or this window appears or even that an error message is shown telling you, the help file were corrupt. Be unconcerned: the help file is completely all right. The Windows help system produces this report automatically if it cannot process a help call correctly. Please, support the development of the TextTransformer and inform under which circumstances the help system failed.

Part XVI

16 References

16.1 References

The TextTransformers owes its origin some publications mentioned below. I have to thank all of the authors.

The TextTransformer is based on the concept for the top down compiler compiler **Coco from P. Rechenberg and H. Mössenböck**, which they have published together in their book: Ein Compiler-Generator für Mikrocomputer, Carl Hanser Verlag München Wien 1988.

H. Mössenböck (ETH Zürich) has written the original version of Coco in Oberon-2. This version was ported to Modula-2 from Marc Brandis (ETH Zürich) and Pat Terry (Rhodes University, Grahamstown, South Africa). Finally Frankie Arzu (Universidad del Valle, Guatemala, Central America) has published a version of Coco in c.
The actual versions of Coco or. Coco/R, which can produce also code in other target languages than c++, can be found at:

http://www.ssw.uni-linz.ac.at/Reserach/Projects/#Coco

The TextTransformer makes use of some **boost libraries**.

http://www.boost.org

These libraries are also required to compile the code which is produced by the TextTransformer.

The source code for the **regular expressions** is **from Dr. John Maddock**.

The **Format library of Samuel Krempp**. It is used in the interpreter for the optional formatting of the output.

The portable functions for the path and file treatment are based on the **filesystem library from Beman Dawes**.

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.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

The parser for the interpreter is based on a **ANTLR** grammar, which can be found at

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.
*/
```

The function table wizard was built by means of the open source wizard component from William Yu Wei, which is part of the Jedi-Vcl :

http://homepages.borland.com/jedi/jvcl/
http://sourceforge.net/projects/jvcl

Xerces-C++ is made available under the 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.

The "ATBinHex" component of Alexey Torgashin was used for the input viewer..

http://atorg.net.ru/delphi/atbinhex.htm

The viewer was extended by the possibility to set breakpoints. The exteded version is available at

http://www.texttransformer.org

The *ATBinHex* is subject to the MOZILLA PUBLIC LICENSE Version 1.1, which is added there too,

My special thanks to **Dr. Hans-Peter Diettrich**, who gave me some valuable notes to improve the program and **Andreas Busch**, who helped me to make TextTransformer multi-user compatible.

Part XVII

17 Glossary

17.1 First set

Each node in the TETRA syntax tree and each symbol of the grammar are characterized by two sets of token: the first set and the follow set. The first set is of special interest for a top down analysis, because it determines the progress of the analysis.

The **first set** of a node contains all token, by which the parser will be guided to the node.
If the actual text isn't matched by one of the token of the first set, an other node will be chosen, namely the node with the first set that contains a token, which matches. If there is no such alternative the parsing is stopped with an error message. To guarantee that the decision between alternatives is definite, the first sets of the alternatives must be disjunctive, i.e.: there may not be a token contained in two alternative first sets.

So the first sets are essential for the progress of the text analysis and the calculation of these sets is the central task of the TextTransformer. To calculate these sets, several cases have to be distinguished:

The simplest case is, a **node of a terminal symbol**. Here the first set exactly contains this symbol.

Example:

The first set of "TETRA" is the set with the Token "TETRA" as its only element.

The first set of a ***concatenation of symbols*** is the first set of its first member (if its not nullable, see below)

Example:

The first set of "TETRA" "means" "Texttransformer" ... is the set with the token "TETRA" as its only element.

More elements are contained in **nodes, which contain alternatives**. The first set of such a node consists in the union of the first sets of the alternatives.

Example:

The first set of ("TETRA" ... | "Texttransformer" ...) is the set of the token "TETRA" and "Texttransformer".

Quite difficult the calculation of first sets of nullable nodes can be. These are the options "(...)? and optional repetitions "(...)*". At first all the tokens belong to the first set, by which the alternative chains inside of the structure can begin. For:

```
( "very"+ | "super" )? "good"
```

this is the set of the token "very" and "super". Because the node is optional, the parsing must

continue, even if "very" or "super" are not found at the actual position. The parsing shall continue with the token "good", or more general, it shall be continued with one of the token, which can follow the nullable structure. The first set of a nullable structure must be united with its follow set: in this case the result is {"very", "super", "good"}. (Again, all set must be disjunctive, see above)

The **follow set** is the set of all the tokens, which can follow a node.

Semantic actions also are presented in the syntax tree as nodes. They are nullable nodes. Their first set exactly is their follow set.

17.2 ASCII/ANSI-set

ASCII is an acronym for "American Standard Code for Information Interchange". The ASCII-code is an assignment of numbers to the characters of the US-American alphabet, to the digits and to the punctuation characters. Furthermore the ASCII-code contains so-called control characters.

In the extended ASCII-code also characters of other languages can be denoted; for example the German umlauts.

Beside of the ASCII-code there are other codes. Windows prefers the ANSI-code (more exact: Windows-1252). For some years has there been the Unicode set which can represent almost all written languages of the world.

Examples:

The upper 'A' has an ASCII-code of 65. The ASCII-code of a space is 32.

You can see the whole table in the TextTransformer: Help->ASCII-table.

Warning: the ASCII-code of a digit is not identical with its numeric value. The digit '2' for example has the ASCII-code 50.

17.3 Backtracking

If the actual information doesn't suffice to decide how to continue the analysis, the possibilities must be tested by a further look-ahead in the text. If such a look-ahead fails, it must be returned to the first state, to test the next possibility. The return to the initial state is called backtracking.

17.4 Binary file

Opposed to a text file the bytes of binary files aren't interpreted necessarily as text characters. E.g. such files are interpreted and represented by special algorithms as pictures or sounds. The documents of word processings are also binary files and complicated algorithms are used to decode the formatting information.

Bytes with the value 0 can occur in binary files. This isn't permitted in text files. This circumstance is used by the example *BinaryCheck*.

17.5 Compiler

A compiler is a program, that translates a higher programming language into a language (binary code) that is understood by the machine.
In a broad sense every translation program of programming languages can be called a compiler. In this sense also the TextTransformer is a compiler, as it compiles the scripts of a project to a text-transforming program.

17.6 Control characters

These characters are used to control the output of the other characters. For example, the line feed character causes that the following characters will be presented in a new line. To denote a control character the backslash is preceded an ordinary character.

17.7 Debug

To debug is the process of finding and eliminating program errors (bugs).

17.8 Deterministic

Grammars are called deterministic, if in each state of the analysis the next step is determined. Deterministic grammars don't need backtracking.

17.9 Escape sequences

The backslash character (\) is used to introduce an escape sequence, which allows the visual representation of certain nongraphic characters. For example, the constant \n is used to the single newline character.
A backslash is used with octal or hexadecimal numbers to represent the ASCII symbol or control code corresponding to that value; for example, '\03' for Ctrl-C or '\x3F' for the question mark. You can use any string of up to three octal or any number of hexadecimal numbers in an escape sequence, provided that the value is within legal range for data type char (0 to 0xff). (For the definition of regular

expression only too hexadecimal numbers are taken.) Larger numbers generate the compiler error "value is too large". For example, the octal number \777 is larger than the maximum value allowed (\377) and will generate an error. The first nonoctal or nonhexadecimal character encountered in an octal or hexadecimal escape sequence marks the end of the sequence.

Take this example.

```
out  <<  "\x0072.1A Simple Operating System";
```

This is intended to be interpreted as \x007 and "2.1A Simple Operating System". However, C++Builder compiles it as the hexadecimal number \x0072 and the literal string ".1A Simple Operating System".
To avoid such problems, rewrite your code like this:

```
out  <<  "\x007" "2.1A Simple Operating System";
```

Ambiguities might also arise if an octal escape sequence is followed by a nonoctal digit. For example, because 8 and 9 are not legal octal digits, the constant \258 would be interpreted as a two-character constant made up of the characters \25 and 8.

All escape sequences which can be used both in literal tokens and in regular expressions are summarized in the following table. There are even further escape sequences for regular expressions.

Note: You must use \\ to represent an ASCII backslash, as used in operating system paths.

Sequence	Value	Char	What it does
\a	0x07	BEL	Audible bell
\b	0x08	BS	Backspace
\f	0x0C	FF	Formfeed
\n	0x0A	LF	Newline (linefeed)
\r	0x0D	CR	Carriage return
\t	0x09	HT	Tab (horizontal)
\v	0x0B	VT	Vertical tab
\\	0x5c	\	Backslash
\'	0x27	'	Single quote (apostrophe)
\"	0x22	"	Double quote
\?	0x3F	?	Question mark
\O		any	O=a string of up to three octal digits
\xH		any	H=a string of hex digits
\XH		any	H=a string of hex digits

17.10 Friedl scheme

The *Friedl scheme* is a pattern, which J.E.F. Friedl formulates in his book: *Regular Expressions*, to avoid *eternal matching*.

Friedl mentions as an example, which is an extremely unhappy case of *eternal matching*:

```
"(\\.|[^"\\\r\n]+)*"
```

This expression defines a string as consisting of the outer double quotes, and an inner repetition. The repetition consists in an alternative of a backslash followed by an arbitrary character (to assure, that the backslash not stands immediately before the closing double quote) and a sequence of characters, which are neither line breaking characters nor a double quote (which would finish the string).
The problem with this expression is, that sequences of characters before a backslash can be interpreted in different manners and that (with a POSIX-NFA regex, as is used in the TextTransformer) all permutations will be tested. Because of the nested repeat-operators '*' and '+', there are extremely much possibilities (three characters are a sequence of one and two characters or a sequence of two and one character). Such a testing (backtracking) leads to a very slow recognition of the whole text. Friedl gives an example, where the evaluation of such an expression exceeds the lifetime of the programmer.

The *eternal matching* can be avoided, if the expression above is reformulated as:

```
"([^"\\\r\n]*(\\.[^"\\\r\n]*)*)"
```

The new expression is subject to the scheme:

*opening normal * (special normal*)* closing*
where the beginning of *special* and *normal* may not be the same

normal forbids a backslash: [^"\\\r\n]
and *special* demands it: \\.
(*opening == closing == ")*

At every location of the text there is a clear alternative now. Indeed, the Friedl scheme is an LL(1)-optimization.

17.11 Interpreter

An interpreter is a program that parses program code and executes it immediately. In contrast to a compiler no storable machine code is produced.

17.12 Lexical analysis

The lexical analysis takes a source text into its elementary pieces that are the lexemes, which are interpreted as token. This process precedes the parsing of the text at least by one step. The parsing analyses the grammatical connections of the token.

17.13 LL(k)-grammar

A grammar is called LL(k) (= from left to right with left canonical derivation and a look-ahead of k symbols deterministic recognizable), if a top down analysis can decide by the next k symbols, how to continue.
Especially a grammar is LL(1), if one token suffices for this decision.

17.14 Numeric systems

We are used to specify integer numbers by 10 different digits. Computers however only know 0 and 1. For example the number **156** of our decimal system expressed in the binary computer format is

10011100

Because such binary numbers can be transformed to octal and hexadecimal representations very easily, the c++ language has conventions for representing octal and hexadecimal values. Leading zeros are used to signal an octal constant and starting a number with "0x" indicates a hexadecimal constant.
The transformation of the binary value above into an octal number is done by separating the binary value in groups of 3 numbers and than replacing these group according to the table below:

10 011 100

2 3 4

In c++ the resulting octal number is written in c++ as: **0234**

The procedure of converting the binary number into a hexadecimal representation is analogously, but you have to separate the digits into groups of four digits this time:

1001 1100

9 C

In c++ the resulting hexadecimal number is written in c++ as: **0x9C**

Escape sequences are a similar representation of characters by octal numbers or hexadecimal numbers.

Binary	Octal
000	0
001	1
010	2
011	3
100	4
101	5

110	6
111	7

Binary	Hexadecimal
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

17.15 Parser

A parser in the narrow sense is a program, which executes the syntactical analysis of a text. In a broader sense this word is used in this help to comprise also the lexical analysis and the execution of the actions.

In the TextTransformer several parser take part. The TextTransformer is a parser generator. So

1. the TextTransformer has a parser to parse the scripts and
2. the TextTransformer creates new parser. The created parser either
   a) exists in the working memory to transform loaded texts, or
   b) exists as a parser class in form of source code

17.16 Parser generator

A parser generator is a program, which creates a parser out of a simple description of a grammar.

17.17 Parse Trees and AST's

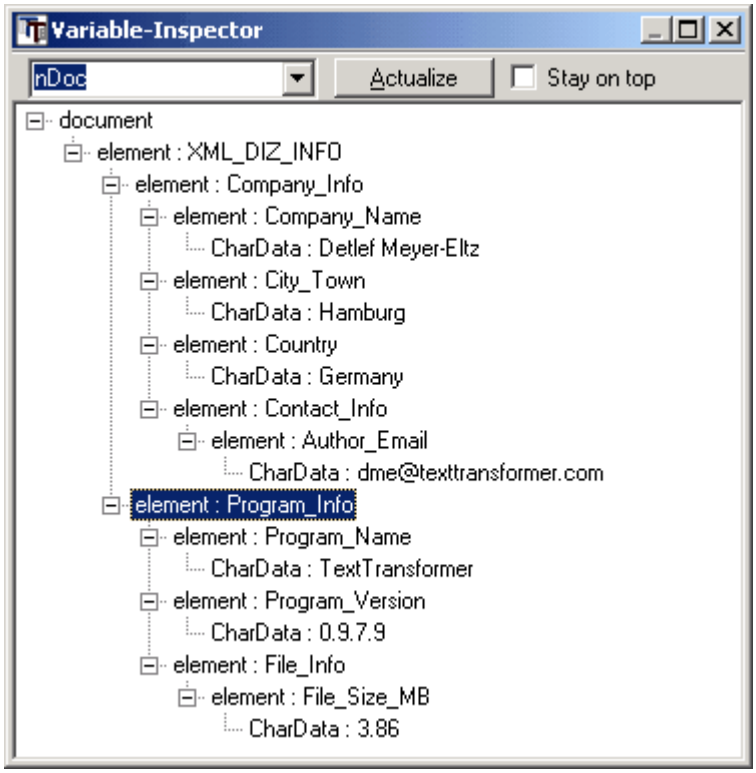
In tree structures the input text can be represented with a structure that conforms to the grammar. Such a tree is called a *parse trees* or *AST (abstract syntax tree).*
The advantages of a previous creation of such parse trees in contrast to an immediate translation

(*one-pass compiler*) is:

- You can make multiple passes over the data without having to re-parse the input.
- You can first perform transformations on the tree itself.
- You can evaluate things in any order you want.

Generally spoken, a tree consists of a number of nodes, connected to each other.

In the following picture a tree structure is depicted, as the variable inspector shows it.



This picture shall demonstrate the relationships of the nodes.

Each node, which has a child, is marked by a little square; the ***leafs*** of the tree don't have such a square. A colon and the value of the node follow the label. All node in the picture have the same label: *element;* only the **root** node at the top of the tree has the label: *document*.

If you pick out the node with the value ***Program_Info*** following relationships exist:

Parent-node is: element : XML_DIZ_INFO
first Child node is : element : Program_Name
last Child node is: element : File_Info
next Sibling node not exists.
previous Sibling node is: element : Company_Info
first Sibling node is: element : Company_Info.
last Sibling node is: element : Program_Info
bottom first Child node is: CharData : TextTransformer

bottom last Child node is : CharData : 3.86

For the **root node**
bottom first Child node is: CharData : Detlef Meyer-Eltz
bottom last Child node is : CharData : 3.86

Walk through a tree

A walk through the tree can be done in **descending or ascending order**. The descending order goes exactly parallel to the line numbers, in which the nodes appear in the picture above. Beginning with the root node:

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

The ascending order is exactly the reversed order.

17.18 Syntax

Syntax is the theory of the structure of a language. A syntactical approach ignores meaning and context of expressions and only looks at their structural relationships.

17.19 Start rule

The start rule is the production, by which the processing of the input begins. Starting from the start rule the matching sub-rules will be called. Therefore the start rule is superior to the other productions of the transformation.

17.20 Text file

For a computer scientist text is a sequence of characters, a character string. The pool of characters a computer can handle is numbered. So a number in the working memory represents a character. The control characters are included in this enumeration.
Which numbers are assigned to which characters, depends on standards for the operating system and the country. The most important standard, also used in the TextTransformer, defines the ASCII set.

In fonts little pictures are assigned to the characters, which represent them on the screen. If in a font there is no picture for a character, the character is represented by a dummy picture: an empty square.

17.21 Top down analysis

The principle of the (deterministic) top down analysis is, to start the analysis of a text with the most general rule - the start rule (top) - and to test, which sub-rule (down) will match the next section of text, and so on.
In contrast to this method, the bottom up analysis begins with the text and looks for a matching rule.

17.22 Token and lexemes

Token are the elements of a text, into which pieces the lexical analysis takes the text. Typical tokens are words, numbers, punctuation characters etc. The tokens of a programming language for example are key words like "double" or "while". In the case of key words, there is a 1:1-relationship between the tokens and the **Lexemes**. A lexeme is a section of text, which represents a token. For example in case of a number there are many lexemes representing the same token; for example: "12", "14.8" or "1001". Such general tokens are described by patterns of text. Inside of the TextTransformer the description of the patterns is done by means of regular expressions.
A problem consists in overlapping tokens. For example: "<" and "<=". The TextTransformer automatically chooses in such conflicting cases the longer lexeme: "<=".

Which parts of a text are tokens, in the end depends on the interpretation of the text.

17.23 Unicode

Unicode is a coding standard that was developed by the Unicode Consortium. Unicode can represent almost all written languages of the world.

However, all the individual characters cannot be represented by a single byte any more as it is the case for the ASCII/ANSI set. There are different methods how the characters can be represented by means of several bytes.

- For the representation of every single character two (or four) bytes are used. This method is used

in the Windows operating system.

- Different characters are coded depending on her general meaning with a different number of bytes. A very common standard, which uses this method, is UTF-8. In the UTF-8 coded Unicode, the first 128 characters of the ASCII code only use one byte. ASCII code and Unicode are identical here. The 128 characters following in the ANSI code are represented in UTF-8 by two bytes each and all further characters need still more bytes for their representation.

Example:

If an UTF-8 coded file is opened in ANSI mode the German word "für" appears as:

fÃ¼r

The German Umlaut 'ü' needs two bytes in the UTF-8 encoding, which are shown as two characters in ANSI mode. But if the file is opened in UTF-8 mode the word is shown correctly.

17.24 Line breaks

Line breaks in text files are represented by special control characters. In the Windows operation systems this is a combination of a carriage return character '\r' and a linefeed character '\n'. If e.g. the single characters of the three-line text:

1. Line
2. Line
3. Line

are listed in a table, this table looks like:

1	.	L	i	n	e	\r	\n
2	.	L	i	n	e	\r	\n
3	.	L	i	n	e		

In UNIX operation systems instead of the '\r\n' combination a single linefeed character is used. The same text there is a little bit shorter:

1	.	L	i	n	e	\n
2	.	L	i	n	e	\n
3	.	L	i	n	e	

In the TextTransformer editor the Windows convention is always used. If UNIX texts are loaded, then a carriage return character is put in front of all linefeed characters automatically. A corresponding warning note is shown than.

In most transformation projects line breaks are ignored. If they are, however, constitutive for the structure of the text to be analyzed, you have to take the additional '\r' characters into account. In the extreme case the exetution of a project then can have different results, depending on whether it happens in the TETRA-working bench or in the transformation manager or with the command line tool, which both treat the unchanged texts.

Both, on Windows and on UNIX, line breaks are recognized by the following token (if the line breaks aren't ignored):

```
EOL  ::= \r?\n
```

Part XVIII

18 Naming conventions

In the help and in the examples some name conventions are used almost generally. They shall simplify reading the scripts. The use of such conventions in your source text also can be of a great use if you want to work on it with the TextTransformer later once. However, the use of the conventions is completely up to you. The use of the TextTransformer depends on it in no way.

Exception: the function table wizard doesn't work correctly with function tables, if the name doesn't begin with "m_ft".

An abbreviated type name is put in front of **variables**.

type	identifier	example
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
function table	ft	ftPrint

m_ is put in front of the names of **class variables**.

Example: m_sColor a class variable of the type str

x is put in front of the names of **parameter variables**.

Example: xsColor a parameter of the type str

Index

- ! -

!: Operator 324
!=: Operator 323

- $ -

$ 248

- % -

% 344
%: Operator 321
%=: Operator 322

- & -

& 56
&&: Operator 324
&: Operator 324
&=: Operator 322

- * -

* 251, 268
*: Operator 321
*=: Operator 322

- . -

. 248, 281

- / -

// 240
/: Operator 321
/=: Operator 322

- - -

-: Operator 321
--: Operator 321

- ? -

? 251, 268
?: Operator 325

- [-

[...] 244, 282

- \ -

\` 248
\< 248
\> 248
\a 248, 442
\b 248
\d 244
\e 442
\f 442
\l 244
\n 328, 442, 450
\r 442, 450
\s 244
\t 442
\u 244
\v 442
\w 141, 244
\z 248

- ^ -

^ 244, 248
^: Operator 324
^=: Operator 322

- _ -

_ 241

- { -

{-...-} 280
{_..._} 280
{{...}} 142, 263, 280

{} 251, 268
{=...=} 280

- | -

| 250, 266
|: Operator 324
||: Operator 324
|=: Operator 322

- ~ -

~: Operator 324

- + -

+ 251, 268
+: Operator 321
++: Operator 321
+=: Operator 322

- < -

<: Operator 323
<< 328
<<: Operator 324
<<=: Operator 322
<=: Operator 323

- = -

=: Operator 322

- - -

-=: Operator 322

- = -

==: Operator 323

- - -

--> 388, 390

- > -

>: Operator 323
>=: Operator 323
>>: Operator 324
>>=: Operator 322

- A -

-a 232
aborting a loop 330
Accept changes in scripts automatically 133
Action 31, 50, 194, 280, 387
add 303, 312
addChildBefore 312
addChildFirst 312
addChildLast 312
AddError 229, 364
AddMessage 229, 364
Addresses 260
AddToken 108, 109, 281, 362
AddWarning 229, 364
aka Latin1 146
Alexey Torgashin 435
Algorithm 253, 373, 374
alnum 244
alpha 244
Alternative 250, 266, 382
Ambiguity 373, 379
Analysis step by step 44
Anchor 248
and 324
Andreas Busch 435
ANSI 122, 441
ANSI set 184
ANSI table 184
ANSI2DOS 165
antlr 435
ANY 271
ANY options 272
Apache 435
append_path 339
Arithmetic operators 321
Arzu 435
ascending order 446

ASCII 146
ASCII set 441
Assignment operators 322
AST 446
Asterix 251, 268
ATBinHex 435
attrib 313
Attribute 313
Auto-save layout 134

- B -

-b 232
back 299
background color of a script editor 188
Backslash 180, 241, 243, 265, 295, 442
Backslash example 296
Backtracking 257, 441
Backup 136, 226, 229
Backup of the project 122
basename 337
BC_CPP 258
BC_PAS 258
BC_XML 258
Beman Dawes 435
bin 329
Binärdatei 145
Binär-Modus 145
binary 358
Binary file 70, 442
Binary files 260
Binary mode 341
Binary number 445
binary open mode 393
Binary output 329
Bitwise operators 324
bjam 397
blank 244
Block commands (editor) 235
BOM 146
bool 293
bool_bin 329
bool_mstrfun 309
boost 253, 410, 435
boost buid 387
Boost Build 397
Boost Software License 435
bottomFirstChild 316
bottomLastChild 316
BranchName 349
break 269, 330
break or continue instruction at invalid position 429
Breakpoint 202, 213, 214
buffer 356
Buffer anchor
  "\A" 248
Buffering the look-ahead tokens 144, 421, 422
Byte order mark 146

- C -

C++ 31, 280, 292, 387
C++ compiler projects 398
C++ header 139
c++ instructions 282
c++ instructions listed 286
Call of a method 284
Calling a parser 393
Calling a production from the interpreter 365
Calling parameters 282
Calling th parser 129
Cannot add branch 430
Cariiage return 442
Carriage return 145, 442
carriage return character 450
Case sensitive 178, 335
Case sensitivity 261, 362
Casesensitive 141, 154
C-Builder 413
Chaining 250
Change start rule 199
change_extension 98, 338
char 294
char_bin 329
CHAR_CPP 257
Character class
  predefined 244
Character class calculator 180
Character type 151, 281
Checking success 211
childCount 313
Circular derivation 415

Circular look-ahead 385
Circularity 380
Circulary look-ahead 416
Class 119
class elements 282, 283
Class members 61
class method 284
class variables 61, 216
clear 295, 299, 303
Clear output 132
Clear semantic code 191
Clear semantic code on all pages 125
clear_indent 359
ClearIndents 359
ClearScopes 361
ClearTokens 362
clock_sec 347
clog 329
clone 312
cntrl 244
Coco 113
Coco/R 98, 99, 113, 272, 277
Cocor 435
Code frame 387
Code generation 270, 387
Col 349
Collapsing semantic code 194
Collating Element Names 246
Collating elements 246
Command line parameter 232
Command line version 232
Comment 239, 264, 285, 370, 375
Comment as inclusion 74
Comments 37, 93, 145, 258
Compiler 442
Compiler compability 410
Completeness 379
Compliment 244
component support 139
ComponentSupport 136
ComponentSupport_Extension 136
Concatenation 266
Conditional operator 325
CONFIG 135
ConfigParam 144, 357
Conflict 375
Conflict detecting 75
conflict resollution 277
Conflict resolution 366
Conflict treatment 76, 269, 380, 384
const 321, 423
const code 151
const parser 152, 306, 355
Constructor example 281
Container 298
Container classes 16
containsKey 303
containsValue 307
continue 330
Control character 295
Control characters 442
Control structures 325
copy 349
Copy code 152
Corrections 230
Corrupt help file 433
cout 328
Cpp_Header_Extension 136
Cpp_ParserHeader 136
Cpp_ParserSource 136
Cpp_Source_Extension 136
Create interface 154
Creating a line parser from an example text 168
Creating a production from an example text 171
CSV wizard 167
ctohs 334
ctos 334
CTT_BufferAbs 404
CTT_BufferAll 404
CTT_BufferBase 404
CTT_BufferLL1ex 404
CTT_DomNode 406
CTT_DynamicScanner 403
CTT_Error 270, 330, 408, 430
CTT_ErrorExpected 408
CTT_ErrorIncomplete 408
CTT_ErrorStandstill 408
CTT_ErrorUnexpected 408
CTT_Exit 408
CTT_Guard 404
CTT_IgnoreScanner 403
CTT_Indent 407

CTT_LiteralScanner 403
CTT_Match 403
CTT_Message 408
CTT_Mstrfun 309, 406
CTT_Mstrstr 405
CTT_Node 406
CTT_NodeError 408
CTT_ParseError 408
CTT_Parser 398
CTT_ParseState 402
CTT_ParseStateDomPlugin 152, 407
CTT_ParseStateDomPluginAbs 407
CTT_ParseStatePlugin 152, 407
CTT_RedirectOutput 407
CTT_RegexScanner 403
CTT_Scanner 403
CTT_SemError 408
CTT_SkipScanner 403
CTT_Token 403
CTT_Tst 403
CTT_TstNode 403
CTT_Warning 408
CTT_Xerces 358, 408
current_path 339
cursor 306, 307
Customize layout 160

- D -

dangling else 382
Data field 260
DATA FOLDER 136
Dates 257
Dawes 435
dbl 349
dbl_mstrfun 309
DD_MM_YYYY 257
Debug mode 44, 134, 155, 208
DebugDefault 208
DebugDefault.ds 155
Debugger 442
Debugging 209
Debugging a look-ahead 100
declaration 55
Decrement 321
Default function 309, 421
Default label 146
Default layout 155
Default return value 263, 404
Default value 198, 285, 293
Delphi 129, 413
Derivability 379, 415
Derivable rules 380
descendentsCount 313
descending order 446
detach_node 312
deterministic 442
digit 244
Digraph 246
Disabling actions 197
Disabling interpreter 197
dnode 119, 146, 152, 311, 320, 393
dnode label 320
dnode_mstrfun 309
do 327
Docking windows 156
document type definition 149
DOM 170, 216, 358, 408
DOMDocument 216
DOS2ANSI 165
Dot 248
double 294
Double quote 442
double_bin 329
Dr John Maddock 410, 435
Dr. Hans-Peter Diettrich 435
Dr. Maddock 253
DTD 149
dtos 333
dummy parameter 422
Dynamic scanner 108, 152, 261, 362

- E -

EBCDIC 146
ebcdic-cp-us 146
EBNF 86, 115
Edit 125
Edit header frame 129
Edit main frame 129
Edit mode 134, 155, 188
Edit source 129

EditDefault.ds 155
Edit-mode 177
Element list 184
else 277, 325
E-mail address 72
empty 295, 307
Empty alternative 266, 381
Enabling actions 197
Enabling interpreter 197
Encoding 122, 145, 146
END 277, 279
End of file 271
endl 59, 328
ends 329
ENTER_CONST_GUARD 404
ENTER_GUARD 404
ENTER_LA_GUARD 404
enums_pas.frm 136, 139
Environment options 135, 136
EOF 102, 260, 271
EPS 266
Equality operator 323
Equivalence classes 246
Error expected 371
Error handling 330, 364, 408
Error message 190, 264, 379, 415
Error on parsing parameters of the parser call 424
Error while parsing parameters 422
Escape sequence 243
Escape sequences 442
eternal matching 444
Euro symbol 146
Events 173, 368
Example GrepUrls 64
Examples 40
exception 330
Exchange of words 41
Excluding individual files 220, 228
Excluding successful transformed files 228
Execute 211
Execution of a project 41
Execution step by step 209
exists 340
EXIT 270
EXIT_GUARD 404
Expected error 371
Expected output 370
expected token 133
Export 122, 194
Extended Backus-Naur form 86
Extender 192
extension 338
Extensions 136
external cursor 306
Extra parameters 220
ExtraParam 144, 357

- F -

F1 415
Failure alternatives for ANY 272
Failure alternatives for SKIP 275
false 293
Family concept 37, 379
Feedback 163
File 122
File extension 136
file filter 137
file mask 137
file_size 70, 340
filesystem 397
filter 137, 220
find 296
find_file 98, 341
find_first_not_of 296
find_first_of 296
find_last_not_of 296
find_last_of 296
findChildId 318
findChildLabel 318
findChildValue 318
Finding 126
findKey 303
findNextId 318
findNextLabel 318
findNextValue 307, 318
findParentId 318
findParentLabel 318
findParentValue 318
findPrevId 318
findPrevLabel 318
findPrevValue 307, 318

findValue 307
First set 202, 203, 440
firstChild 316
firstSibling 316
float_bin 329
Folder structure 223
follow 316
Follow set 202, 203, 273, 440
F-Option 275
for 326
format 344
format example 65
Formatting 342
Formfeed 442
Frame directory 136
Frame for component support 413
Frame path 139
Frames 136
Friedl scheme 257, 444
front 299
function table 309
Function table example 105
Function-Table-Wizard 174

- G -

Generate c++ code 129
GenError 364
getCursor 299, 303, 306
GetDocumentElement 320, 358
GetMsgType 364
GetState 349
GetUseExcept 364
Global scanner 142, 154, 403
Go to the actual position 218
gotoNext 307
gotoPrev 307
Grammar 374
Grammar test 379
graph 244
greedy 271
GrepUrls example 64
Group 370
Grouping 250, 267

- H -

hasAttrib 313
hasChildren 313
hasCurrent 307
HasError 364
HasMessage 364
Header frame 388
Header/Chapters/Footer wizard 169
header-only 397
Help directory 136
Help file 433
Help system 433
HEX_CPP 256
HEX_PAS 256
hexadecimal 70
Hexadecimal code 47
Hexadecimal escape sequence 243
Hexadecimal number 243, 445
hexadecimal numbers 349, 442
Highlighting during look-ahead 210
hstoi 332
Html-Tags 251

- I -

ibm037 146
ibm1047 146
ibm1140 146
Icon of a node 200
id 254, 313
if 70, 277, 325
IF structure 197
Ignorable characters 93, 139, 259, 416
IGNORABLE_PAS 259
IGNORE_CPP 259
IGNORE_XML 259
Ignored characters 154
Implementation frame 390
Implicit xState parameter 284
Import 111, 113, 122, 192
Include directories 97
Include files 96, 98
Inclusion 74, 375, 421, 422
Inclusion does not exist 416

Inclusions 35, 37, 93, 145
Incomplete parse 420
incr_indent 359
Increment 321
IncrIndent 359
indent 359
indent_str 359
Indentation 359
Indentations in the generated code 152
IndentStr 359
Info 219
Ini file 313
Initialization 285, 431
InitProcDeclaration 388
InitProcImplementation 390
Input 370
Input window 121, 196
Instructions for the interpreter 282
instructions of c++ 286
int 256, 294
int_bin 329
int_mstrfun 309
Interactivity 197, 198
Interaktivity 154
Interface method 154
InterfaceDeclarations 388
InterfaceImplementations 390
internal cursor 306
Interpreter 31, 110, 280, 444
Interpreter instructions 282
invalid cursor 306
is_directory 98, 340
isAncestor 313
isDescendant 313
IsLastFile 356
IslastFile example 66
ISO-8859-1 146
ISO-EBNF 86
isSibling 313
IsSubCall 349
isValid 307
itg 349
itohs 334
itos 333

- J -

jamfile 387, 397
Java 98
Jedi-Vcl 435

- K -

key 303
Keyboard shotcuts 234
Krempp 435

- L -

la_copy 349
la_length 349
la_str 349, 366
label 313
la-buffer 144, 421, 422
Language 135
lastChild 316
LastPosition 349
lastSibling 316
LastSym 349
Layout 134
LC 258
Left recursion 150, 384
length 295, 349
level 313
Level of look-ahead 210, 215
Lexem 449
Lexical analyser 373
Lexical analysis 30, 373, 444, 449
Library for regular expressions 410, 435
License 410
Line 349
Line anchor 248
Line begin 248
Line break 259, 341, 450
Line breaks 152
Line comment 240
Line end 248
Line feed 442
Linefeed 145, 328, 442
linefeed character 450

list of all instructions 286
Literal 177, 178, 241
literals 241
LL(1) analysis 379
LL(1) conflict 75, 277, 366
LL(1) principle 35
LL(1) test 380
LL(k) grammar 445
Load layout 161
load_file 98, 341
load_file_binary 341
LoadFike 393
Loading data 118
Local options 93, 116, 154
Local scanner 142, 403
local variables 216
locale 245, 246
log 329
Log file analyzis 168
Log window 121, 219
Log-file 222
Logical operators 324
Look ahead 277
Look ahead example 99, 100
Look ahead finishing 270
Look-ahead 35, 197, 210, 281, 366, 385, 416
look-ahead debugging 100
Look-ahead example 70
Look-ahead level 215
Look-ahead stack 150
Look-ahead token buffer 144
Look-ahead, no variable inspector 216
lower 244
lp_copy 349
lp_length 349
lp_str 349

- M -

-m 232
Macro 252
Maddock 410, 435
main 393
Main parser 35
Management 67, 112, 219, 232
Management.ttp 112
Mangement 231
map 303
Mark recognized/expected token 212
Marked token 133
mask 220
Match 373
match_results 403
matched 349
Matching but not accepted token 417
MemberInitialization 390
memory leaks 362
Meta character 264
Meta characters 243
Methods of CTT_Parser 399
MIME-parser 72
Minimal distance for character ranges 180
Mismatch between declaration and use of parameters 422
Modulo 321
Mössenböck 435
MsgBegin 364
MsgEnd 364
mstrbool 303
mstrchar 303
mstrdbl 303
mstrdnode 303
mstrfun 309
mstrint 303
mstrnode 303
mstrstr 303
mstruint 303
Multiline regular expression 252
Multiple replacement of characters wizard 165
Multiple replacement of strings wizard 165
Multiple replacement of words wizard 164
Multithreading 151, 152
Multi-user compatibility 130

- N -

N:1 222
N:1 example 67
N:1: Transformation 227
N:N 222
N:N Transformation 223
Name 239, 262, 283, 369

Name of a node 200
Named literal 241
Named literal tokens 129
Named literals 241
Navigation 189
New project wizard 122, 163
Newline (linefeed) 442
next 316
Next token 209
next_copy 349
next_length 349
next_size 349
next_str 349
nextLeaf 316
nextSibling 316
NFA-Option 275
NF-Option 275
No failure alternatives for ANY 272
No failure alternatives for SKIP 275
no help 433
node 119, 200, 311, 312
Node breakpoint 214
node/dnode differences 312
node::npos 316
node_mstrfun 309
Non circularity 379
Non-breaking space 244
Non-circularity 380
Nongraphic characters 442
Non-terminal 379, 380
Non-terminal symbol 30, 33, 380
Nonterminalsymbol 32
not 324
npos 296, 316
NULL 70, 260
Nullability 381, 415
Nullable structure 382
Nullable structure in a repetition 416
Numbers 256
Numeric systems 445

- O -

Octal number 445
octal numbers 349, 442
OK 270
Oktal number 243
OnAcceptToken 368
OnBeginBranch 368
OnBeginDocument 368
OnEndBranch 368
OnEndDocument 368
OnEnterProduction 368
one-pass compiler 446
OnErrorExpected 408
OnErrorIncomplete 408
OnErrorStandstill 408
OnErrorUnexpected 408
OnExitProduction 368
Only code for initializations is allowed hereI 431
OnParseError 349, 368
Open 122
Open mode 145, 393
Operating system 152
Operators 321
Option 251
Options 130
or 324
Ostream 393
out 328, 358
Output 328
output buffer 356
Output in a second file 119
Output window 121
Overlap 431

- P -

-p 232
Paramerter 284
Parameter 282
Parameter and {{...}} 142
Parameter declaration 239, 263
Parameter field 56, 263
Parameter wizard 172
parent 316
parse 346
Parse all connected scripts 189
Parse all scripts 190
Parse single script 189
Parse start rule 189
Parse system 271

Parse Tree 86, 110, 174, 446
Parse tree example 102
Parser 374, 446
Parser call 129
Parser class 387, 388, 390
Parser generator 30, 446
Parser interface 349
Parser system 375
ParserClassName 388, 390
ParserHeaderName 390
ParserHeaderSentinel 388
ParserRuleDeclarations 388
ParserRules 390
ParseTree 311
Parse-tree 170
Pascal 96
PATH 136
path_separator 342
Pipe character 250, 266
Placeholder token 108, 261, 281, 362
Plugin method 152, 355
Plugin type 152
Plugin-type 393
Plugin-variables 216
Plus 251, 268
pop_back 299
pop_indent 359
PopIndent 359
PopScope 109, 281, 361
Popup menu 202
Position 349
POSIX 253
Predefined character classes 180
Predefined tokens 253
Predicate 277
Preprocessor 71, 138
Pretty-print 146
prev 316
Preview of the target files 228
prevLeaf 316
prevSibling 316
print 244
printf 342, 344
Processing instructions 93
Production 30, 33, 110, 262, 265
Production as function 33
Production as look-ahead 366
Production list 184
Production system 375
ProductionName 349
PROGRAM FOLDER 136
Project directory 136
Project frame 396
Project menu 154
Project options 137
punct 244
push_back 299
push_indent 359
PushIndent 359
Pushscope 109, 281, 361

- Q -

Question mark 251, 268, 442
Quick wizard for function tables 176
Quotation mark 265
Quote 178
Quotes 257
Quoting escape 243

- R -

-r 232
random 348
Reachable 379
REAL 256
Rechenberg 435
Reckognized
but not acceptet token 425
Recognized token 133, 215
Redirection 119
Redirection of the output 358
RedirectOutput 358
RedirectOutputBinary 358
Redo 125
Referenz 56
Refresh 228
Regex Test 80, 178
Registration 18, 432
Regular expression 31, 178, 242
multiline 252
Relational operators 323

remove 299, 303
Remove all scripts 191
removeChild 312
Repeat 251, 268
replace 295
replaceChild 312
Replacing 126
Report transformation results 230
Repository 379
reset 212, 299, 303
ResetOutput 358
Restoring default Debug layout 162
Restoring default editing layout 162
Results 229
return 330
Return type 239, 263, 404
Return value 263, 404
return value of productions 59
returning a value 330
RFC 822 72
rfind 296
Roll back 231
root 316
Root label 146
RTF 51
Rule 33, 265

- S -

-s 232
Samuel Krempp 435
Save 122
Save layout 161
Scaffolding for semantic actions 172
Scanner 373
ScannerEnum 388
Scope 109, 361, 427
ScopeStr 361
Script 238
  accept changes 188
  cancel changes 188
  delete 188
  edit 188
  insert 187
  rename 189
Search window 126
Searching 126
Section of text 197
Select source files 220
Select target directory 223
Semantic action 240
semantic actions 282
Semantic code 194
Semantical action 387
Set target to source directory 223
setAttrib 313
SetDefaultLabel 406
setId 313
SetIndenter 359
setLabel 313
SetPosition 349, 362
Sets 244
SetState 349
Setting pattern for the target files 226
Settings folder 130, 162
setValue 299, 303, 313
Single quote 442
size 295, 307, 346, 349
SKIP 178, 203, 215, 248, 273, 384
SKIP node with SKIP neighbours 384, 415
Skip options 275
SKIP text 336
SKIP token matches at actual position 417
sortCildrenD 320
sortCildrentA 320
source code 398
source folder 398
Source text 196, 197
Source window 121
SourceName 356
SourceName example 65
SourceRoot 356
SourceRoot example 65
space 244
SQL 94
stack 150, 301
Stack size maximum 150, 424
Stack window 215
standstill 426, 430
Standstill in look ahead 426
Star 268
Start 211

Start a transformation 229
Start and successor of nullable structures 382, 415
Start mode 208
Start of several alternatives 382, 415
Start parameter 144
Start position 132, 197
Start rule 37, 138, 198, 271, 379, 448
StartRule 390
StartRuleDeclaration 388
StartRuleHeading 390
State 349, 354
status bar color 211
stay on top 216
std::string 295
std::vector 299
std::wstring 295
Step into 209
Step over 209
stoc 332
stod 85, 331, 349
stoi 332, 349
str 295, 346, 349
str method of the plugin 358
str() 356
str::npos 296
str_mstrfun 309
string 257, 295
string_bin 329
struct 119
Structure 119
Sub parser 365
Sub-expression 178, 250, 354
Sub-parser 35, 37, 98, 118
substr 295
Supporting code 397
switch 327
Symbol number 215
Symbolic name 243
Symbolic names 246
Syntactical analysis 373, 446
Syntax 448
Syntax highlighting 309, 379
Syntax tree 32, 184, 200
  presentation 133
Syntaxbaum 32
System 375
System overlap 431

- T -

-t 232
Tab 442
Tabulator 442
Target 136
Target text 328
Target window 121, 328
TargetName 356
TargetRoot 356
Template parameter for plugin character type 153
temporary file 226, 229
Terminal symbol 30, 31, 380
Terminalsymbol 32
Ternary operator 325
Test 38
Test all connected scripts 189
Test all literals 142
Test all scripts 190
Test definition 370
Test list 184
Test output 371
Test script 369
Test single script 189
Testing of all literals 116
TETRA 16, 30
tetra folder 398
TETRA script language 33, 110
tetra_cl.exe 232
TetraComponents 129, 368, 413
Text 349, 449
Text breakpoint 213
Text file 70
Text mode 341
Text scope 108, 362
Text section 197
Text-Modus 145
Text-scope 361
TextTransformer 16, 30
this 216
throw 330, 430
time_stamp 347
to_lower_copy 335
to_upper_copy 335

Token 30, 31, 32, 238, 265, 379, 449
Token definition 240
Token lis 184
Token set 374, 375
Token text 240
Token window 215
TokenList 390
Tool bar 121, 186
Toolbar 121
Topdown analysis 445, 449
Torgashin 435
Transformation manager 219
Transformation manager example 67
Transformation of groups of files 219
Transformation options 222
Transformations-Manager 16
Transitional action 281
Tree wizard 173
Tree-Wizard 173
trim_copy 336
trim_left_copy 335
trim_right_copy 336
true 293
tte 192
tti 192
ttm 231
tto 137
ttparser_c.frm 136, 139
ttparser_h.frm 136, 139
ttr 192
ttt 192
ttx 192
TTXercesLib 410
Type 284
Type conversion 331, 332, 333, 429
Typedef 108

- U -

UCS4 146
uint_bin 329
uint_mstrfun 309
Underline 241
Undo 125
Unexpected symbol 418
Unicode 86, 151, 449
Unix 152
Unknown identifier 427
Unknown symbol 415
unsigned int 294
Upgrade 18
upper 244
URI_ANGLE_DELIM 254
URI_QUOTE_DELIM 254
URI_WS_DELIM 254
UseExcept 364
User data 130, 136
User error 430
UTF-16 146
UTF-8 86, 122, 146, 358, 449

- V -

value 307, 313
Value is too large 442
Variable 431
variable declaration 55
Variable types 293
Variable-Inspector 16, 216
Vasant Raj 260
vbool 299
vchar 299
vdbl 299
vector 299
Version number 153
Versions 16
Video 40, 155, 167, 173
Viewer 196
vint 299
Virtual methods 399
visit 309
Visual Express C++ 404
vnode 299
vstr 299
vuint 299

- W -

Warning 190, 379, 381, 415
wchar_t 151
while 279, 326, 327
WHILE structure 197

Wide character 151
Window list 159
Windows 152
Windows menu 155
Windows-1252 146, 441
Wizard for a Header/Chapters/Footer frame 169
Wizard for a new project 122
Wizard for creating a line parser from an example text 168
wizard for CSV files 167
Wizard for multiple replacement of characters 165
Wizard for multiple replacement of strings 165
Wizard for multiple replacement of words 164
Wizard for the creation of a production from an example text 171
Wizards 163
word 244
Word anchor 248
Word begin 248
Word boundaries 261
Word boundary 178
Word bounds 141
Word end 248
word processing 442
WORD_EN 255
WORD_FR 255
WORD_GE 255
Words 255
wregex 151
WriteDocument 320, 358
wstring 151, 295

- X -

xdigit 244
Xerces 320, 358, 397, 408
Xerces (license) 435
Xerces portability 410
xercesdom folder 398
XercesInclude 393
XercesInit 393
XercesLib 410
XercesUsingNamespace 393
XML 86, 320
xState 349, 354
xState parameter 284
xState.str(index) 250

- Y -

Yacc 384
Yu Wei 435
YYYY_MM_DD 257

- Z -

Zeilenumbruch 145