PROSPECTRA
an ESPRIT Project

Manual for the ELL(2)-Parser Generator and Tree Generator Generator

Reinhold Heckmann

Universität des Saarlandes
6600 Saarbrücken
Bundesrepublik Deutschland

PROSPECTRA Deliverable Item S.1.1-R-2.1

1986-08-26

Distribution: public

ABSTRACT

Regular right part grammars extended by tree generator specifications are interpreted by a combined parser generator and tree generator generator that produces an ELL(2) parser. This parser is able to translate programs of the specified language into abstract syntax trees according to the tree specifications in the generator input.

Public

© 1986 by

Reinhold Heckmann
Universität des Saarlandes

in the Project

PROgram development by
SPECification and
TRAnsformation

sponsored by the

Commission of the European Communities

under the

European
Strategic
Programme for
Research and development in
Information
Technology

Project Ref. No. 390

Introduction

This manual consists of four main parts: a system overview, the description of the generator input (a regular right part grammar extended by tree specifications), the description of actions and output of the generator (combined parser generator and tree generator generator), and the description of the generated parsers (combined ELL(2) parsers and abstract syntax tree generators).

Some aspects of the generator are therefore considered twice, once when explaining the input, once when describing actions and output.

1. System overview

1.1. Principal design of the input grammar

A simple parser that can be generated from a context-free grammar, is able to parse a sentence of the language defined by the grammar into a (concrete) syntax tree. Such trees contain unimportant terminals like parentheses, semicolons etc. used only to structure the original linear sentence, and non-terminal chains resulting from productions having only one non-terminal on the right hand side.

Abstract syntax trees do not contain such superfluous informations, but on the other hand they may include nodes standing for things not occurring in the string form, e.g. the abstract syntax tree for

if B then S1 else S2 could be

and for *if B then S*

in order to respect the ternarity of the node marked by *'if-stat'*. If we want to generate a parser directly constructing abstract syntax trees, we must therefore extend the context-free grammar to control the tree construction process.

The generator input consists of some specifications, of a list of terminal symbols, and of the extended grammar itself. The grammar contains a list of productions where regular expressions are produced by the non-terminal symbols. A regular expression is built by concatenations and alternations from some atomic elements.

The atomic elements are:

- Non-terminal symbols
- Unimportant terminals only occurring in the string
- Important terminals occurring in string and tree
- Operator names only occurring in the tree like *'null'* in the example above
- The word *EMPTY* occurring neither in the string nor in the tree; it may occur in the grammar to improve its readability
- List descriptions standing for star expressions

1.2. Implementation

Generator and generated objects are implemented in the programming language Pascal on a VAX 11/780 computer of Digital Equipment Corporation under operation system Unix 4.2 bsd, a trademark of Bell Laboratories.

There are three deviations from Standard Pascal in the system:

1) The Unix operation system allows the user for giving an arbitrary number of arguments separated by

blanks in addition to the call of a program. These command line arguments may be read by Unix Pascal programs using some predefined variables and procedures. In our system, command line arguments serve as options to control the activities of both generator and generated object.

2) *reset* and *rewrite* may be called with two parameters to achieve the association of internal Pascal files with external Unix files.

3) Unix Pascal programs may contain lines
   #include "*file*"
   Such lines are replaced by the contents of the given file when the Pascal compiler reads the source code. This include facility is used only in the generated object, not in the generator.

1.3. Interaction of system components

The parts of the system and their relations are figured in the picture. Rectangles denote executable programs and ovals data or source code files. The depicted relations are maximal, i.e. it is possible to let the generator output only a parser table and nothing else, or only an Optran tree grammar.

Optran [2] is a tool for transformations of attributed syntax trees. The generated objects may be used as Optran front-end as figured in the picture, or they may be used for other purposes or in their own right.

The generator is controlled by options given as command line arguments, and reads its input file from standard input. This input contains lexical, syntactic and tree generation specifications. The generator prints listings and messages onto standard output, a parser table derived from the syntactic specifications, a scanner generator input that is essentially a copy of the lexical specifications, and an Optran tree grammar derived from the tree generation specifications.

POCO [1] is a compiler generator system here used as scanner generator. POCO takes as command line arguments no options, but names of input and output files. It reads from the first file, gives listings and messages on the second file, and generates the scanner onto the third file.

The extractor is a little program written in the C language that extracts a slightly transformed scanner table from the POCO generated scanner. This is necessary since the POCO scanner driver is also modified to allow the integration in the generated object.

The decfile module was written by B. Weisgerber; its purpose is to construct the so-called decfile for Optran.

In fact, there are four different parser drivers: *notrpas.i* for a parser written in **Pas**cal that generates **no** **tr**ees, *treepas.i* for a **tree** generating parser, *muenpas.i* for a **mu**lti-**en**try tree generating parser, and *incrpas.i* for an **incr**emental tree generating parser. The scanner driver is on file *scan.i* and the decfile module on file *decfile.i*.

All these different parts of the generated object are combined by the include facility and translated by the standard Unix Pascal compilers *pc* or *pi*.

Assume the specification for language *lan* is on file lan.spec. Then the appropriate sequence of commands to generate a complete front-end for the language *lan* is as follows:

```
generator c:lan s+ o+ (other options) < lan.spec > lan.messages
POCO lan.scspec lan.scan.messages lan.scan
extract < lan.scan > lan.scan.i
pc lan.p
```

This will produce a file of messages of the generator (lan.messages) and of POCO (lan.scan.messages) and an Optran tree grammar (lan.optranprog). The characters '<' and '>' bind Unix file names to standard input and output; the other files are bound by options to the generator.

The generated object also accepts options from the command line, and reads the program to be parsed from standard input. On standard output, it gives listings, messages and the abstract syntax tree in readable form. If it is used as front-end for Optran, it also generates the tree in Optran code and additional informations for Optran about scanner attributes etc. in the program that was parsed.

2. The generator input

2.1. Overview

The generator input is given to the generator as standard input. It consists of six sections:

options section
classes section
terminals section
typename section
axiom section
productions section

The sections must appear in this order, but some sections are not obligatory and may be missing. The *options* section allows for giving options together with the input, *classes*, *terminals*, and *typename* section define the lexical structure, and *axiom* and *productions* section the syntax of the language to be analyzed by the generated parser.

The command line is considered as first input line, thus its contents are treated as ordinary input. Normally, the command line will contain only options, but it would also be possible to place a complete input grammar on it.

2.2. Options

The *options* section may be missing. If present, it consists of an arbitrary number of options separated by blanks, tabs, or newlines.

An option consists of an option specifier (a lower case letter), a value (a single character, in most cases **'-'** for 'disabled', **'+'** for 'enabled', and **'#'** for 'strongly enabled'), and in some cases an argument (a number or a word). Blanks, tabs, and newlines are option separators, thus an option must not contain them, but between two options, there must be at least one of them.

If an option is not given or given without argument, a default value resp. a standard argument is assumed. For some options, this default depends on other options (see below). The order of options is arbitrary, but if two options with the same code are encountered, the second one is ignored. Since the command line is evaluated before the ordinary input, it is possible to override options given inside a generator input file by options on the command line.

Listing **l-** **l+** Default: l+
Controls the listing of the generator input. If l- is given on the command line, the listing is totally suppressed.

Messages **m-** **m+** Default: m+
If m- is given, only warning and error messages are output; otherwise also positive messages are given.

Informations **i-** **i+** **i#** Default: i-
i+ : Output the grammar graph in readable form
i# : Output grammar graph together with first- and follow-sets
The output of these informations is the last action of the generator. If an error occurs during the reading of the language specification, there is no complete grammar graph and thus no output of informations about it.

ELL(2) checking **e-** **e+** **e#** Default: e#
e- : no computation of first- and follow-sets and no ELL checking
e+ : computation of first- and follow-sets and ELL(2) checking
e# : like e+, but with automatic ELL(2) conflict resolving (see 3.4)

Printing messages about ELL(1) conflicts **p-** **p+** Default: p+
If p- is given, only messages about ELL(2) conflicts are output.

Tree condition checking **t-** **t+** **t#**
Default: if gn then t- else if o+ or o# then t# else t+
The tree conditions are statements about the completeness and consistence of tree generation

specifications (see 2.7, 3.5). The default value of this option depends on the values of the options *g* and *o*.

t- : none of the tree conditions is checked
t+ : conditions (TC1) through (TC3) are checked, but not the arity condition (TC4)
t# : all four tree conditions are checked

Generation **g- gn gt gm gi**
Default: if e- then g- else if t- then gn else gi

g- : Don't generate a parser
gn : Generate a pure parser without tree generation
gt : Generate a string-to-tree parser
gm : Generate a multi-entry string-to-tree parser
gi : Generate an incremental string-to-tree parser

Without ELL(2) checking, nothing can be generated; without checking of tree conditions, no parser with tree generation can be generated. If the given value of the *g* option is "greater" than the default (in the order $g- < gn < gt < gm < gi$), an error occurs.

Code file name **c:***cname* Default: c:code

If the given name *cname* has suffix '.p', this suffix is omitted by the option reader. With c:*cname*, the generated parser table is printed onto file *cname*.p, and the scanner table is expected to be on file *cname*.scan.i . *cname* is also used as prefix for the default values of other file names.

Scanner specification: **s- s+ s+***sname* **s# s#***sname*
Default: s- , *sname*=*cname*.scspec

The default *sname*=*cname*.scspec is chosen, if s+ or s# without following *sname* are given.

s- : Don't output the scanner specification
s+*sname* : Output scanner specification onto file *sname* and continue processing of the input
s#*sname* : Only output the scanner specification and do nothing else

Option s# is appropriate if an error was detected and corrected only in the lexical specification part of the input i.e. parser table and Optran tree grammar need not be recomputed. s# is more powerful than all other options, i.e. gi, e#, t#, etc. are not respected if s# is given.

Optran tree grammar: **o- o+ o+***oname* **o# o#***oname*
Default: o- , *oname*=*oname*.optranprog

o- : Don't output a tree grammar
o+*oname* : Output simple tree grammar onto file *oname*
o#*oname* : Output better tree grammar onto file *oname*

Optran tree grammars are very restricted, thus only an approximation of the real tree grammar *G*" of chapter 2.7 can be output. This approximation specifies a superset of the real tree language of the generated tree generator. The approximation of o+ does not contain anything of the original structure and respects only the arity of operators, but is quickly computed; whereas the tree grammar of o# is as close as possible at the real tree grammar, but hard to compute. See chapter 3.6 for further details.

Attributes: **a- a+** Default: a+

If a+ is given, the Optran tree grammar output in case of o+ or o# is attributed by scanner attributes declared as "local" and "imported". These scanner attributes contain information about place and kind of terminal symbol instances found by the scanner.

"Recursive" instead of "iterative" trees: **r- r+ r+***prefix*
Default: if o+ or o# then r+empty_ else r- ; r+ means r+empty_

Actual Optran does not accept a special kind of trees ("iterative trees"). These must be replaced by corresponding "recursive" trees if the generated object shall be used as front-end for Optran. The exact explanation and the meaning of the prefix are given in chapter 2.7.

Format options:

fm*number* Default: fm80
fg*number* Default: fg80
fo*number* Default: fo80

The number gives the maximum line length of the messages file (fm), the generated code (fc), and the Optran tree grammar (fo). The generator breaks lines such that this length is not exceeded. If a number less than 60 is given, it cannot be granted that all lines respect this maximum length. The listing of the input is not formatted, but directly copied from input onto output.

2.3. Lexical considerations

2.3.1. Tokens in the generator input

The generator input is a sequence of "important" tokens separated by blanks, tabs, newlines, or comments. In the following chapters, we introduce some token classes that may occur in different sections; token classes specific for one section are defined when the section is described.

2.3.2. Comments

Comments are sequences of characters starting with **(*** and ending at the first subsequent occurrence of ***)**. Inside a comment all characters including newline are possible. Comments may be given everywhere between the other tokens.

2.3.3. Symbol and operator names

A symbol name is a non-empty sequence of freely usable characters; its end is given by the first subsequent character that is not freely usable. An operator name is a sequence of freely usable characters enclosed by quotes ′. These quotes are part of the operator name, thus it is impossible that an operator name equals any symbol name.

Freely usable characters are the letters 'a' through 'z' and 'A' through 'Z', the digits '0' through '9', and the special characters

! @ # $ % ^ & _ - + = ~ ` \ ; " < > , ?

The generator distinguishes between upper and lower case letters.

Names of terminal symbols should not begin with '+', '-', '=', or '~'; otherwise the *terminals* section reader could be disturbed. Examples for legal symbol names are

statement STATEMENT Statement <statement> "statement"
if-statement s @ !s+7 11 &

Note that all these symbol names are different from each other.

Although symbol and operator names may be arbitrarily long, only their first 32 characters are stored in the generator. Thus two names are considered as equal if they do not differ in their first 32 characters, and the deparser does not output the whole name if it is longer than this limit.

Symbol names may be used as names for terminal and non-terminal symbols, and operator names denote operators (markings of nodes in the abstract syntax tree). Terminal and non-terminal symbols are distinguished by looking up in the list of terminal symbols given in the *terminals* section. Thus it is impossible to use the same name for a terminal and non-terminal symbol.

2.3.4. Reserved keywords

Some possible names are reserved as section headers to structure the input. They are

CLASSES	*TERMINALS*	*TYPENAME*	*AXIOM*
classes	*terminals*	*typename*	*axiom*

They are not reserved inside the *productions* section, thus they may be used as non-terminal names, but not as terminal names.

In addition, the names

PRODUCTIONS	*FINIS*	*EMPTY*	*ROOT*	*LEFT*	*RIGHT*	*EPS*
productions	*finis*	*empty*	*root*	*left*	*right*	

are reserved to have special meaning only in the *productions* section. Thus they may be used neither as terminal nor as non-terminal names.

Corresponding upper and lower case keywords have the same meaning for the generator, they may be freely mixed. Names like *Empty*, < *empty* >, or "*empty*" are not considered as keyword and may be freely used. The word *EPS* is reserved since it denotes the empty word ε in the output of the generator.

2.4. *classes* and *terminals* section

These sections declare the terminal symbols of the grammar and contain the lexical specifications that are handed over to the scanner generator POCO. Thus their structure is closely related to the structure of POCO input.

The *classes* section may be omitted and consists of the section header *CLASSES* or *classes* followed by POCO character class definitions, e.g.

$$letter = 'A' - 'Z', 'a' - 'z';$$

These character class definitions are ignored if option s- is given, and copied to the file *sname* without change otherwise.

Warning:

Character class names must be POCO names i.e. contain only letters and digits (first character must be a letter), and must be different from the POCO reserved names.

The *terminals* section may also be omitted and consists of the section header *TERMINALS* or *terminals* followed by a list of terminal symbol declarations. The structure of this declarations is

symbol-class-mode   symbol-class-name   lexical-specification

If no scanner generation is intended (i.e. with option s-), lexical specification and symbol class mode are ignored and may be missing. With option s+ or s#, they are obligatory.

To understand the different symbol class modes note that a parser considers terminal symbols as atomic units without further structure whereas a scanner considers them as classes (regular sets) of several instances. When a POCO generated scanner detects an instance of a terminal symbol in the input stream, it submits four numbers to the calling parser: class code (number of class the instance belongs to), relative code (number of instance in the class), line number, and column number. The parser uses only the class code for its purposes, the other three numbers are stored somewhere to be used by eventual semantic analysis.

There are different methods to compute the relative code; the appropriate method is chosen by the user by means of the class mode. It may be '+', '$*$', '-', or ε (invisible).

ε: This mode is only allowed for finite classes consisting of some fixed instances. The scanner generator associates a relative code with each instance, and the generated scanner cannot alter this association. An example is the class of multiplication operators in Pascal consisting of the five instances $*$ / *div mod and*.

'+': This mode is appropriate for infinite classes where each instance shall uniquely be associated with a relative code when it is first encountered by the scanner. Subsequent occurrences of the same instance lead to the same relative code as the first occurrence. Typically, identifiers in a programming language are defined to form such a class.

'$*$': This is another possible mode for infinite classes. All occurrences of instances are numbered from 1 on by the scanner such that different occurrences of the same instance lead to different numbers. This is more efficient than '+' mode since the scanner need not compare new instances with previous ones. The class of strings in programming languages may be given this mode since it is likely not necessary to know whether two strings are equal.

’-’: Instances of this class are ignorable, e.g. comments or sequences of blanks, and will not be passed on to the calling parser.

Hint: It is recommendable but not obliged to place the declarations of ignorable terminal symbols at the end of the declaration list. Then the parser generator does not count them, does not do work for them, and does not generate space and code for them; thus generator and generated parser work more efficiently than otherwise.

The lexical specification for a class is either structured as in POCO:

= *regular-expression* ;

or consists only of the single character ’~’. This character may be used if a terminal class shall contain only one instance that equals the terminal class name (typical for keywords in programming languages).

Terminal symbol names are not subject to the POCO restrictions since the parser generator renames them automatically when it outputs the POCO input. The original names are added as comments such that the user be able to understand eventual POCO error messages.

A disadvantage of this renaming is that occurrences of terminal symbol names inside a regular expression belonging to another terminal symbol, must be marked by an exclamation-point ’!’ to allow for their renaming. Besides this exception, the regular expression is formed as described in the POCO manual.

Remark: It is necessary to separate the terminal symbol name from the following ’=’ or ’~’ character of the lexical specification by space since ’=’ and ’~’ might be part of a symbol name. Symbol names occurring after ’!’ inside a lexical specification need not be separated from their context (see also 3.1.5).

As an example we give the *classes* and *terminals* section for a small language and the scanner specification output by our generator.

```
classes

    letter              = ′A′ - ′Z′, ′a′ - ′z′;
    digit               = ′0′ - ′9′;

terminals

∗   integer             = digit ∗ [digit];
∗   real                = ! integer ′.′ !integer;
+   identifier          = letter ∗ [(letter | digit)];
    while               ~
    do                  ~
    od                  ~
    read                ~
    write\ln            = ( ′write′ | ′writeln′ );
    add-op              = ( ′+′ | ′-′ );
    mul-op              = ( ′∗′ | ′/′ | ′div′ | ′mod′ );
    rel-op              = ( ′<′ | ′<′ ′=′ | ′=′ | ′<′ ′>′ );
                                  (∗ ′<=′ or ′<″=′ would not work ∗)
-   comment             = ′(′ ′∗′ allbut ( ′∗′ ′)′ );
-   space               = ∗1-[′ ′];
```

This is transformed into:

```
(∗$S+,C+,Q+∗)
TERMINALS

    letter              = ′A′ - ′Z′, ′a′ - ′z′;
```

```
    digit                    = '0' - '9';

 1   *  t001   (* integer      *) = digit  * [digit];
 2   *  t002   (* real         *) = t001 (* integer *) '.' t001 (* integer *);
 3   +  t003   (* identifier   *) = letter  * [(letter | digit)];
 4      t004   (* while        *) = 'while';
 5      t005   (* do           *) = 'do';
 6      t006   (* od           *) = 'od';
 7      t007   (* read         *) = 'read';
 8      t008   (* write\ln     *) = ( 'write' | 'writeln' );
 9      t009   (* add-op       *) = ( '+' | '-' );
10      t010   (* mul-op       *) = ( '*' | '/' | 'div' | 'mod' );
11      t011   (* rel-op       *) = ( '<' | '<' '=' | '=' | '<' '>' );
(* '<=' or '<''=' would not work *)
12   -  t012   (* comment      *) = '(' '*' allbut ( '*' ')' );
13   -  t013   (* space        *) = *1-[' '];

AXIOM  S
PRODUCTIONS
  S: .
FINIS
```

As you see, there are the following transformations:

1) A header and a tail are added since POCO must be controlled by some options and expects at least one production.
2) The keyword *classes* is replaced by *terminals*, and *terminals* is omitted.
3) The terminal declarations are numbered.
4) The symbol names are replaced by t001 etc. In our example, the names *if, then, else, while, do, add-op, mul-op, rel-op,* and *write\ln* would have been forbidden without renaming.
5) The tilde '~' is expanded into a regular expression.
6) The regular expressions are copied without change except renaming forced by exclamation-points.

2.5. *typename* section

This section is ignored and may be missing if option g-, gn, or gt is given, but is oligatory in case of gm and gi. Then the text to be parsed by the generated parser may contain "holes" i.e. so-called "type names" derived from non-terminal names standing for a subconstruct left unspecified. The set of type names must be disjoint to the set of instances of normal terminal symbol classes, and must be lexically specified as '+' class somewhere in the *terminals* section. The *typename* section states this terminal name and defines how the type names are to be derived from the non-terminal names.

The *typename* section may have one of the following structures:

typename *terminal-symbol-name*
typename *terminal-symbol-name* firstchar *character*
typename *terminal-symbol-name* lastchar *character*
typename *terminal-symbol-name* firstchar *character* lastchar *character*

where *typename*, *firstchar*, and *lastchar* are keywords that may also be given upper case.

Some examples will show the meaning of this section. Assume you want to specify a grammar for a simple language whose only sentence is *id*, *id* where *id* is standing for identifiers being strings of letters.

Consider the following grammar without *typename* section:

CLASSES

```
    letter              =  'A' - 'Z', 'a' - 'z';

TERMINALS
+   id                  =  *1-[letter];
    comma               =  ',';
-   space               =  *1-[' '];

PRODUCTIONS
    Pair                :  First  comma  Second.
    First               :  id.
    Second              :  id.
FINIS
```

(This grammar and the other ones in this paragraph are incorrect since the tree generation specifications are omitted for reasons of simplicity.)

With this grammar, it is impossible to generate a parser that also accepts sentences where subconstructs are left unspecified. If you want this, you must choose type names standing for these "holes". Taking the non-terminal names immediately as type names is impossible since the names *First* and *Second* are instances of the class *id*. One possible solution is to take @*First* and @*Second* as type names:

```
CLASSES
    letter              =  'A' - 'Z', 'a' - 'z';

TERMINALS
+   id                  =  *1-[letter];
    comma               =  ',';
+   types               =  '@' !id;
-   space               =  *1-[' '];

TYPENAME  types  FIRSTCHAR  @

PRODUCTIONS
    Pair                :  First  comma  Second.
    First               :  id.
    Second              :  id.
FINIS
```

The parser generated from this grammar accepts the completely qualified sentence *id*, *id* as well as the sentences with "holes"

@Pair
@First, @Second
@First, *id*
id, @Second

Another possible solution is to use non-terminal names that are not instances of *id*:

```
CLASSES
    letter              =  'A' - 'Z', 'a' - 'z';

TERMINALS
+   id                  =  *1-[letter];
    comma               =  ',';
+   types               =  '<' !id '>';
-   space               =  *1-[' '];

TYPENAME  types
```

```
PRODUCTIONS
    <Pair>          :  <First>  comma  <Second>.
    <First>         :  id.
    <Second>        :  id.
FINIS
```

Accepted sentences: *id*, *id*   <First>, *id*   etc.

A completely equivalent parser would be generated from this grammar:

```
CLASSES
    letter          =  'A' - 'Z', 'a' - 'z';

TERMINALS
+   id              =  *1-[letter];
    comma           =  ',';
+   types           =  '<' !id '>';
-   space           =  *1-[' '];

TYPENAME  types  FIRSTCHAR  <  LASTCHAR  >

PRODUCTIONS
    Pair            :  First  comma  Second.
    First           :  id.
    Second          :  id.
FINIS
```

Now we give the exact correspondence between non-terminal and type names:

Let wf and wl be two strings such that

$wf = wl = \varepsilon$		for	TYPENAME T
$wf =$ 'c',	$wl = \varepsilon$	for	TYPENAME T FIRSTCHAR c
$wf = \varepsilon$,	$wl =$ 'c'	for	TYPENAME T LASTCHAR c
$wf =$ 'c_1',	$wl =$ 'c_2'	for	TYPENAME T FIRSTCHAR c_1 LASTCHAR c_2

Then the type name belonging to the non-terminal N is the concatenation of wf, N, and wl.

Remarks:

1) There must not be blanks between wf and N or N and wl in type names.
2) The type name symbol class must have mode '+'. It is recommended but not obliged to place it at the end of the *terminals* section just before the first ignorable class.
3) The type name symbol class must not be used as terminal symbol in the productions of the grammar.

2.6. *axiom* and *productions* section: the input grammar

2.6.1. The *axiom* section

The *axiom* section defines the axiom of the grammar. It looks like

AXIOM *non-terminal*

The *axiom* section may be omitted, the axiom is then the non-terminal on the left hand side of the first production.

2.6.2. Tokens in the *productions* section

Key characters

Key characters are colon **:**, dot **.**, the bracket pairs **()[]{}**, and finally **|** and **/**.

Iteration operators

There is a non-empty iteration operator which is .. or equivalent ... and an empty iteration operator, that is ..0.. or also ...0... (the character among the dots is the numeral zero). Use and meaning of this operator and of the other characters with special meanings are described in the next chapter.

2.6.3. Syntax of the *productions* section

The *productions* section has the following structure:

PRODUCTIONS
sequence of productions
FINIS

Each production consists of a non-terminal (left hand side) followed by a colon **':'** ("produces") and a regular expression (right hand side), and is completed by a dot **'.'** . A regular expression is a sequence of regular terms separated by an alternation symbol; that is **'|'** or equivalent **'/'**. A regular term is a sequence of regular factors that may be empty.

There are seven kinds of regular factors:

1) regular expressions enclosed in parentheses ( ) ;
2) the keyword *EMPTY* (see below);
3) non-terminal symbols;
4) pure terminal symbols (unimportant terminals that will not be put into the syntax tree);
5) node descriptions defining leaves of the syntax tree;
6) node descriptions preceeded by keyword *ROOT* specifying the root of the subtree corresponding to the actual production;
7) list descriptions.

A node description is either an operator or an operator followed by [*terminal*]. The described node is marked by the given operator, and in the second case, the given terminal is consumed from the input of the parser and attached to the described node as scanner attribute.

The simplest form of a list description is {*operator*}. It defines an empty list also occurring in the tree. Other legal forms of list descriptions are

{	N	T'	it	N' }		(only allowed if no trees shall be generated)
{ O	N	T'	it	N' }		
{	N O [	T]	it'	N' }	A	
{	N O		it'	N' }	A	

where

O is an operator name

N is a non-terminal

N' is either empty or a repetition of non-terminal N

T is a terminal

T' is either a terminal or empty

it is an arbitrary iteration operator

it' is the iteration operator .. or ...

A is an associativity specification that is either *LEFT* or *RIGHT*.

The keyword *EMPTY* is standing for the empty string. It may be omitted, but in some situations, it improves the readability of the grammar.

Examples:

- *if Condition then Statement-list* (*else Statement-list* | *EMPTY*) *fi*
  is equivalent to
  if Condition then Statement-list (*else Statement-list* |) *fi*

- An empty production may be written as *A*: *EMPTY*. or as *A*: . .

A list description must not contain the keyword *EMPTY*; if a part is said to be empty, this means it is the empty string and thus invisible.

The informal syntactic meaning of these list descriptions is "list of N separated by T resp. T'" e.g. {*Statement semicolon ... Statement*} is standing for "list of statements separated by semicolon".

Symbol names and the different kinds of iteration operators are described in the previous chapter. The keywords may be given in upper or lower case letters. The meaning of the various syntactic constructs with respect to the definition of a language and the specification of tree generation is given below.

A correct grammar has to satisfy some context conditions:

(CC1) Each non-terminal must have at most one production.

(CC2) Each non-terminal must have at least one production.

(CC3) Each non-terminal must be productive, i.e. able to derive at least one string of terminal symbols.

(CC4) Each non-terminal must be reachable from the axiom.

(CC5) In list descriptions, the second non-terminal must equal the first one.

There are some further conditions with respect to the completeness and consistence of tree generation specifications. These tree conditions and the ELL(2) conditions that guarantee the existence of an ELL(2) parser for the input grammar, are given below.

2.6.4. Limitations to the input grammar

The length of input lines is restricted to 255 characters. The number of terminal symbols is limited by constant "tzahl" actually being 255, the number of non-terminals by constant "nzahl" that is 1000, and the number of operator names by constant "ozahl" being 2000. Huge regular expressions having hundreds of nesting levels may cause the stack of the grammar reader to overflow. The maximum stack length is given by the constant "maxparse" whose current value is 1000. The maximum number of regular factors in a regular term and of regular terms in a regular expression is 255.

2.7. Meaning of the input grammar

2.7.1. Introduction

The input grammar G is a regular right part grammar extended by tree generation specifications. Thus it has two meanings: it defines a context-free language $L = L(G)$ and a mapping $T = T(G)$ associating an abstract syntax tree with each sentence in L.

We shall define the correspondence between G, L, and T by use of an auxiliary context-free grammar G' that has only strings on the right hand sides of its productions, but in general has an infinite number of productions, and a tree grammar G'' that is closely related to G'. G' has the same set of terminal symbols as G, and the language $L = L(G)$ belonging to G is defined as $L = L(G')$.

The tree grammar G'' has the same set of non-terminal symbols as G', and as many productions as G'. The axiom of G'' is a single non-terminal leave S where S is the axiom of G'. Each production p': $X_o \rightarrow X_1 \cdots X_k$ of G' corresponds to a production p'': $X_o \rightarrow t(p'')$ where $t(p'')$ is a tree that has as many occurrences of non-terminal leaves as there are occurrences of non-terminal symbols in $X_1 \cdots X_k$

such that there is a one-to-one correspondence between them.

All nodes in $t(p'')$ that are not non-terminal leaves - called real nodes from now on - are marked by an operator. Some of the real nodes are attributed by a terminal symbol that is related to an occurrence of this terminal symbol in $X_1 \cdots X_k$. The scanner attributes of these occurrences of terminal symbols will be attached to their related attributed nodes when parsing a concrete sentence of L. Those real nodes that are not attributed are called pure nodes.

The abstract syntax tree $T(w)$ of a sentence w in $L = L(G')$ with given G'-derivation path is obtained as last element of a G''-derivation path starting at the axiom of G'' and using corresponding productions of G'' at corresponding non-terminal leaves.

For an example, assume that grammar G' among others has the productions

Expression $\rightarrow$ *Term*,
Term $\rightarrow$ *Factor1 mulop Factor2*,
Factor $\rightarrow$ *identifier*,

where *mulop* and *identifier* are terminal symbols. Let the corresponding tree productions be

The numbers after *Factor* are given to define the required correspondence between occurrences of non-terminals in p' and p''.

The derivation of the abstract syntax tree of $a * b$ then looks like

2.7.2. Operators

Before we give the construction of G' and G'' from G, we must consider the different kinds of nodes on the right hand sides of tree productions. There are non-terminal leaves that we simply denote by the name of the non-terminal. The other nodes are marked by a string called operator like $'mulop'$ or $'id'$ in the example above.

An operator is said to be of fixed arity if the generator guarantees that each node in the tree productions marked by this operator has the same number of children. The other operators are called list operators. In pictures, we represent nodes with operators of fixed arity as circles or ovals, and list nodes - nodes with list operators - as rectangles. In our example, $'mulop'$ is an operator of fixed arity 2 and $'id'$ of 0.

Actual optran does not allow for list operators; all operators must have fixed arity. If a generated parser shall be used as front-end for Optran, the iterative construct of list nodes must be transformed into a recursive replacement:

is replaced by

In the recursive tree, there is an additional operator, ′empty_op′ in our example. This operator does not occur in the generator input and is constructed by the generator if option r+ is given. Its name is derived from the name of the corresponding list operator by the prefix given in the r+ option (default is *empty_* as in our example). A parser generated with option r+ is able to construct both iterative and recursive trees according to a run-time decision of the user, whereas a parser generated with r- is only able to construct iterative trees since the additional prefixed operators are missing.

2.7.3. Construction of G' and G'' from G

2.7.3.1. Ordinary productions

Let $G = (V_N, V_T, P, S)$ be the input grammar. V_N denotes its set of non-terminal symbols, V_T its set of terminal symbols, P the production set, and S the axiom. Then let $G' = (V'_N, V'_T, P', S')$ and $G'' = (V''_N, V''_T, P'', S'')$ where $V''_N = V'_N = V_N \cup \{n(ld) \mid ld$ is a list description occurring somewhere on the right hand sides of the productions of $G\}$, $V'_T = V_T$, $S'' = S' = S$. V''_T is the set of real nodes on the right hand sides of the productions of G''.

Each production N: R. from P leads to a set of some productions in P' and P''. The regular expression R consists of an alternation of concatenations of regular factors. Regular factors may be

(RE)	regular expressions enclosed in parentheses '(' ')'	
(EM)	the keyword	*EMPTY*
(NT)	non-terminal symbols:	*non-terminal*
(PT)	pure terminal symbols:	*terminal*
(PO)	pure operators:	*operator*
(AO)	attributed operators:	*operator* [*terminal*]
(PR)	pure root-statements:	*ROOT operator*

(AR) attributed root-statements: *ROOT operator* [*terminal*]
(LD) list descriptions.

The first step to obtain productions of G' and G'' from a production of G, is to transform the regular expression R using the distributive law into an alternation of c concatenations of regular factors of kind (EM) through (LD). Each concatenation will be refined to the right hand side of a production of G' and a corresponding production of G''. Thus the original production of G leads to c productions of G' and G''.

Let $rf_1 \cdots rf_n$ be such a concatenation of regular factors of kind (EM) through (LD). Let $\#xx$ be the number of factors of kind (XX) in it (e.g. $\#pt$ = number of pure terminals in it). Thus $\#re = 0$ and $\#em + \cdots + \#ld = n$ holds. To obtain the right hand side of a production of G' from this concatenation, delete all *EMPTY*'s, pure root-statements and pure operators from it, thus only factors of kind (NT), (PT), (AO), (AR), and (LD) are remaining. Then replace attributed operators and root-statements by the terminal contained in it, and replace the list descriptions ld by the non-terminals $n(ld)$. The resulting string is the right hand side of a production of G' and consists of $\#pt + \#ao + \#ar$ terminal and $\#nt + \#ld$ non-terminal symbols.

To obtain the tree in the corresponding tree production, delete at first all *EMPTY*'s and pure terminals from the concatenation $rf_1 \cdots rf_n$, then extract all root-statements out of it. Let $m = \#nt + \#po + \#ao + \#ld$ be the length of the resulting concatenation $rf'_1 \cdots rf'_m$. The tree production can be constructed only if the input grammar satisfies some tree conditions:

(TC1) There is at most one root-statement in the original concatenation $rf_1 \cdots rf_n$: $\#pr + \#ar \leq 1$.

(TC2) There is at least one regular factor in the refined concatenation $rf'_1 \cdots rf'_m$: $m \geq 1$.

(TC3) If there is more than one regular factor in the refined concatenation, there must be a root-statement in the original one:
$m > 1 \rightarrow \#pr + \#ar > 0$.

There is one further tree condition given below.

The right hand side of the tree production then looks as follows:

$k1 \cdots km$ are nodes corresponding to the regular factors $rf'_1 \cdots rf'_m$ according to the following table:

The required correspondences between the $\#nt + \#ld$ occurrences of non-terminal symbols and between the $\#ao + \#ar$ attributed nodes resp. important terminals in the production of G' resp. G'' are given by the left-to-right order in the original concatenation $rf_1 \cdots rf_n$.

2.7.3.2. Productions resulting from list descriptions

From a list description, we in general construct an infinite number of productions of *G*' and corresponding productions of *G*". In the following table, we write these productions by use of a parameter *k*. *'op'* denotes an operator name, *N* a non-terminal name, and *T* a terminal name. *N1, N2,* $\cdots$ denote occurrences of *N* and *T1, T2,* $\cdots$ occurrences of *T* to define the correspondence between *G*' and *G*". The dots "..." in the list description are real syntactic dots (iteration operator); the dots "$\cdots$" in the productions of *G*' and *G*" are meta-dots abbreviating a sequence of similar units. We first give the list description, then in the same line the right hand sides of the corresponding productions of *G*'. The trees on the right hand sides of the associated productions of *G*" are figured below; if some different list descriptions lead to the same family of trees, it is drawn only once to save space.

{'op'}	ε
{'op' N ... N}	N1 $\cdots$ Nk , k > 0
{'op' N ..0.. N}	N1 $\cdots$ Nk , k $\geq$ 0
{'op' N T ... N}	N1 T1 $\cdots$ Nk-1 Tk-1 Nk , k > 0
{'op' N T ..0.. N}	N1 T1 $\cdots$ Nk-1 Tk-1 Nk , k $\geq$ 0

With option r-, list nodes are generated:

With option r+, the list nodes are avoided. For the example, we assume the standard prefix *empty_*.

If the generated parser shall not generate trees, the operator name may be omitted in the four kinds of list description above.

{ N ′op′ ... N } left	N1 $\cdots$ Nk , k > 0

{ N ′op′ [T] ... N} left	N1 T1 $\cdots$ Nk-1 Tk-1 Nk , k > 0

{ N ′op′ ... N } right N1 $\cdots$ Nk , k > 0

{ N ′op′ [T] ... N} right N1 T1 $\cdots$ Nk-1 Tk-1 Nk , k > 0

There are equivalent forms of list descriptions: "..." and ". ." are equivalent; ". . 0. ." and ". . . 0. . .", too; the second non-terminal in the list description may be omitted, e.g. {*′op′ N ... N*} is equivalent to {*′op′ N* ...} and {*N ′op′* [*T*] ... *N*} *LEFT* to {*N ′op′* [*T*] ...} *LEFT*.

One further tree condition is

(TC4) Operators of fixed arity must have the same number of children in all productions of *G*".

This condition and the determination of arity are explained in detail in chapter 3.5.

2.7.3.3. Example

We now give an input grammar for a part of a language as example where the keywords *ROOT*, *EMPTY*, *LEFT* etc. are written in upper case, terminal names in lower case, and non-terminal names are capitalized.

```
CLASSES

    letter          = ′A′ - ′Z′, ′a′ - ′z′;
    digit           = ′0′ - ′9′;
```

```
TERMINALS

    identifier        = letter  * [(letter | digit)];
    number            = *1-[digit];
    while             ˜
    do                ˜
    od                ˜
    if                ˜
    then              ˜
    else              ˜
    fi                ˜
    left-par          = ′(′;
    right-par         = ′)′;
    semicolon         = ′;′;
    comma             = ′,′;
    assign            = ′:′ ′=′;
    relop             = ( ′=′ | ′<′ | ′>′ | ′<′ ′=′ | ′>′ ′=′ | ′<′ ′>′ );
    mulop             = ( ′*′ | ′/′ );
    addop             = ( ′+′ | ′-′ );

AXIOM               Statement-list

PRODUCTIONS

Statement-list:     {′statement-list′ Statement semicolon ..0.. Statement}.

Statement:          while Expression do Statement-list od                 ROOT ′while-stat′
                    | if Expression  then  Statement-list                  ROOT ′if-stat′
                         ( ′null′ | else Statement-list )
                      fi
                    | ′var′ [identifier]  assign  Expression               ROOT ′assign′
                    | Call
                    | ′null′.

Call:               ′proc′ [identifier]                                    ROOT ′call′
                       ( {′paralist′}
                       | left-par {′paralist′ Expression comma ... Expression} right-par ).

Expression:         Simplex  (  |  ROOT ′relop′ [relop]  Simplex ).

Simplex:            {Term   ′addop′ [addop] ... Term} LEFT.

Term:               {Factor ′mulop′ [mulop] ... Factor} LEFT.

Factor:               ′id′ [identifier]
                    | ′const′ [number]
                    | ROOT ′sign′ [addop]   Factor
                    | left-par Expression right-par.

FINIS
```

Using this grammar, the tree corresponding to the sentence

while a * b = c do x := - (x + y) * z; p (x + y + 1); q od

looks like

3. Actions and output of the generator

The generator consists of eight parts activated in the following order: input reader; reduction tester; first- and follow-set computer; ELL(2) checker; tree condition checker; Optran tree grammar generator; code generator; grammar graph printer.

The first two parts cause the generator to exit if they detect fatal errors. The two generating parts may be suppressed by options and also by error flags set by previous parts. The following chapters contain a detailed description of the various parts and their interactions.

3.1. The input reader

The input reader consists of a base system reading input lines and placing error messages, section readers - one for each section - , and a scheduler activating the appropriate section reader when a section header is encountered.

3.1.1. Base system of the input reader

The base system reads input lines and sends them to the various section readers. In the other direction, it gets error messages coming from other parts of the input reader and places them into the input listing.

Listing of the actual input line is delayed until either an error is detected in it, or a new input line is to be read and option l+ is given.

When a line is listed, it is preceeded by a line number and tabs are expanded into the appropriate number of blanks. The command line is numbered by zero. The listing of input lines is independent of the format options since there is no additional breaking of long input lines.

Error messages force the actual input line to be listed even with option l-. This listing is followed by a line only containing the character '^' pointing to the place where the error was detected. In rare cases, an error message is not related to the actual input line, but to a previous one; the base system knows about this and then places the pointer at the beginning of the actual line. After the line with the pointer, an error diagnosis follows that is formatted to fit into the maximum line length defined by the "fm" option.

As in ordinary compilers, these error messages must be considered with caution. The place where an error is detected might be somewhere after the place where the real error occurred, and the error diagnosis might be too general or misleading. The problem is that the generator cannot divine what the user intended.

3.1.2. The scheduler

The scheduler activates the section readers at the appropriate time. At first, the options reader is activated to read the command line and the beginning of the input. *options*, *classes*, and *terminals* section essentially consist of a repetition of similar units. Thus the corresponding section readers - once activated - read input until they find a section header. Then they return control to the scheduler that considers the section header and activates the appropriate section reader, or emits the error message "Error: Illegal section" if the section header encountered belongs to a section that must occur before the section just left.

The *typename* section has finite length, and thus the typename reader returns control to the scheduler not only if it finds the next section header. Thus it is possible that the scheduler gets control from the typename reader and does not find a valid section on the input. Then the message "Error: Illegal section" is issued, too.

Assume as example the user wanted to specify a "first character" in the *typename* section and has forgotten the character itself. The result will be

```
50          typename t firstchar
51          axiom s
            ^
+++++ Error: Illegal section
```

What is going on here? The typename reader takes the letter 'a' of *axiom* as "first character", considers the next input word that is now *xiom*, recognizes that this word is not the *typename* section keyword *lastchar*, concludes that the *typename* section is over, and returns control to the scheduler. The scheduler also considers the word *xiom*, recognizes that it is not among the section headers and emits the error message. The pointer '^' is below the letter 'a' since nobody has consumed the erroneous word *xiom* and thus the last consumed character is the letter 'a'.

axiom and *productions* section are both read by the grammar reader that is active until *finis* is encountered. Then the task of reading input is finished.

If an error was detected by a section reader, the scheduler does not activate the next reader but causes the program to exit. Execution is also terminated if option *s#* was selected and the *terminals* section is over. If *axiom* or *productions* section are encountered, there was no *typename* section, and the *g* option has value *gm* or *gi*, then the message "Error: "TYPENAME" section missing" is emitted. If the end of input is encountered, there was no *productions* section, and *s#* was not chosen, the message "Error: "PRODUCTIONS" section missing" is printed.

The described flow of control implies that grave disorder will result from misspelling section headers or using section headers erroneously as e.g. terminal names since then a section reader will make the attempt to read a piece of input that is not understandable for it.

3.1.3. The options reader

The task of the options reader is to consume the *options* section, to interpret the found options, and to check some of them for consistence against each other. The tokens recognized by the options reader are space delimited words. Only their first 32 characters are interpreted, the remainder is cut off silently.

The options reader is always activated as first section reader since the *options* section has no header. Naturally, the first word of input is skipped since this word is the name of the generator program itself (remember that the command line is considered as part of the input).

When interpreting the found words as options, the options reader may detect some errors and then emit messages that are listed together with the actual input line as described above. The pointer '^' then points to the last character of the erroneous word.

We now give a list of sample bad options and corresponding error messages that are written beside the bad options to save space in this document.

-l 123 L+	Error: Option must start with lower case letter
x+	Error: No such option
t@ l# g+	Error: Invalid argument in option
c+code	Error: Form of option "c" is "c:NAME"
fx90	Error: No such format option
fm70a	Error: Invalid number
fm1000	Error: Number too large
fm59	Warning: Number too small
fm39	Warning: Number too small. It is set to 40

When the *options* section is over, the options reader checks the options for consistence. Error messages produced then are not related to the actual input line and thus not introduced by a pointer '^'. The various messages are

Error:	Options "g*C*" and "e-" are inconsistent	*where C ∈ {n, t, m, i}*
Error:	Options "g*C*" and "t-" are inconsistent	*where C ∈ {t, m, i}*
Error:	Options "o+" and "r-" are inconsistent due to the actual implementation of Optran *(same with o#)*	

Error: With option "o+" or "o#", "t#" is necessary

3.1.4. The classes reader

The task of the classes reader is to consume the *classes* section and to copy it onto the scanner specification file if s+ or s# is chosen.

The classes reader consumes its input character by character only looking for a new section header. It does not worry about tokens or syntax and thus it is not able to detect errors. Errors in the *classes* section will thus not be detected before the scanner specification is read by POCO.

3.1.5. The terminals reader

The task of the terminals reader is to consume the *terminals* section, to construct a list of terminal symbols for later use in the grammar reader, and with option s+ or s#, to output the *terminals* section onto the scanner specification file after having done the transformations described in chapter 2.4. Together with the list of terminals, it stores the information whether a terminal symbol is ignorable or not. It may detect two kinds of error when constructing this list:

Error: Terminal declared twice
Error: Keyword used as terminal symbol

The second message is emitted if keywords like *root* that are not section headers are used as names of terminal symbols.

There is an internal limit (255) for the number of terminal symbols. Exceeding it causes the message "Stopped. Number of terminal symbols is greater than constant "tzahl"" and immediate exit of the generator program.

If option s+ or s# was selected and the lexical specification of a terminal symbol class has been forgotten, the generator gives a warning and adds an empty specification "= ;" to the scanner specification file. This kind of error cannot be detected before the next terminal symbol is encountered, therefore the message is coming too late:

```
20        begin
21        end   ~
          ^
```

+++++ Lexical specification missing

When reading the regular expression forming the lexical specification of a terminal symbol class, the reader does not worry about syntax or meaning of this expression; it is only looking for semicolons - indicating the end of the expression - and exclamation-points - marking a word for renaming. It only distinguishes quoted from free occurrences of these symbols to be able to correctly treat declarations like

semicolon = ';';

If a free occurrence of '!' is encountered, the reader reads the next space delimited word and looks up for it in the terminal list created so far. If the word is not found there, its last character is omitted and it is searched again. This is repeated until a search succeeded or until the word has shrunk up to ε. In the latter case, the message "Warning: Renaming forced by '!' failed" is printed.

The last two error messages are only warnings since they do not affect the work of the parser generator. But the corresponding POCO input is then erroneous. Even if the parser generator has not claimed any error or warning, the POCO input may be erroneous since *classes* and *terminals* reader do not perform the POCO syntax analysis and don't look for POCO specific errors.

If lexical specification errors in these two sections are corrected, there is no need for a complete run of the parser generator and tree generator generator, use option *s#*. Complete new generation is only required if names and order of terminal symbols up to the last ignorable one are changed.

3.1.6. The typename reader

The task of this part is to read the *typename* section and to store the defined typename, "first" and "last" character somewhere. It is able to detect three kinds of errors concerning the typename:

Error: Keyword used as typename
Error: Ignorable terminal used as typename
Error: Typename not declared as terminal

It does not check that the typename was really declared as '+' symbol class since the terminals reader does not provide information about the symbol class except the distinction ignorable - not ignorable.

3.1.7. The grammar reader

The task of the grammar reader is to read the *axiom* and *productions* section and to construct an internal representation of them, the so-called grammar graph. Therefore it consists of a scanner, a parser, and a grammar graph constructor working in parallel. Whereas the previous section readers are written by hand and not divided into scanner and parser, the grammar reader was partly generated because of its greater complexity. The scanner was generated by POCO, and the parser by an earlier version of the generator itself from a ELL(1) grammar.

The task of the scanner is to partition the input stream into lexical tokens and to identify the token class. It uses the terminal list constructed by the terminals reader to distinguish between terminal and non-terminal names. Characters that cannot belong to any token class are indicated as "Illegal symbol" by the scanner. Because of the great lexical variety of the tokens, only control (not directly printable) characters may be illegal symbols. It is possible that they are not visible in the listing such that the error message might be looking mysteriously.

If the parser has detected a syntax error, it prints a header essentially containing the word "Error", then a list of symbols that it would expect at the error position followed by the word "expected". Unfortunately, the set of terminals the grammar reader claims to expect is often different from the set of symbols which are really legal continuations of the input in the error situation. At the error position, the parser switches into panic mode and consumes symbols until a dot marking the end of the actual production is encountered.

In some cases, the parser is expecting only one symbol forming a singleton symbol class. The parser then makes an attempt to correct the error by inserting just this uniquely defined expected symbol. The message is then "Error: ... inserted". Such errors are not considered as fatal and do not cause the generator to exit after reading is terminated. But the user should then be careful and check the insertion for correctness.

The parser accepts a wider class of list descriptions than really allowed. This is done to achieve more specific, semantics oriented error messages. It follows a list of sample incorrect list descriptions and the corresponding error message (without pointer ’^’ to save space).

{}	Error:	There must be something between the braces
{′op′} **left**	Error:	The description of an empty list must not be followed by an associativity specification
{**′op1′** N **′op2′** ... }	Error:	Two operators at one list description
{**′op′** N ... } **left**	Error:	Such a list must not be specified as left or right associative
{N **′op′** [T] ..**0**.. } left	Error:	Separating operator not allowed if list could be empty
{N **′op′** [T] ... }	Error:	List must be specified as left or right associative if separating operator is given
{N T ... }	Error:	Missing operator at list description
	(only error with t+ or t#)	
{′op′ **N1** ... **N2**}	Warning:	Second non-terminal in list description should equal the first one
	The second one is ignored	
{N **root** ′op′ [T] ...} left	Warning:	"ROOT" not allowed in list description
	The word "ROOT" is ignored	

There are some more error messages of the grammar reader:

Error: Non-terminal has more than one production
Error: Typename used as terminal symbol
Error: Terminal is ignorable
Error: Implicitly constructed operator *O* was already defined

The last message is printed if option r+*prefix* was selected, a list description {′op′ N (T) ..(0)..} is encountered, and the operator ′*prefix*op′ to be constructed has already occurred directly in the input grammar. The grammar reader does not check whether there is a direct occurrence of the constructed operator after the list description where it was constructed. This asymmetry is caused by technical reasons. The later occurrence may be detected by the tree conditions checker or may even be legal if used in a context with arity 0.

Some of these errors don’t cause the generator to exit after reading has finished, but disallow parser generation.

Large grammars or huge regular expressions may cause the grammar reader to overflow. The emitted error messages are

Stopped. Number of non-terminal symbols is greater than constant "nzahl"
Stopped. Number of operators is greater than constant "ozahl"
Stopped. Stack overflow in grammar reader

To overcome the first two errors, the number of non-terminals resp. operators has to be decreased. The third message is issued if the actual regular expression is nested too deeply. The implicit constructed operators need storage, too, thus a grammar may be accepted under option r-, but not under r+.

Superfluous pairs of parentheses are automatically removed by the grammar reader, e.g. $A\ (B\ C)$ is transformed into $A\ B\ C$, $A \mid (B\ C) \mid (D \mid E)$ into $A \mid B\ C \mid D \mid E$. This is normally not recognized by the user, but may be important to understand error messages produced by a deparser from the grammar graph.

If the grammar reader terminates, it gives a summary of its work:

1) If there were no errors, and option m+ was chosen: "No syntax errors"

2) If there were only syntax errors corrected by insertion: "The grammar reader inserted *n* symbols. Please check these insertions for correctness"

3) If there were other syntax errors or some of the other semantic errors listed above: "Execution stopped because of syntax errors or fatal semantic errors"

Execution is continued only in the first two cases.

3.2. The reduction checker

The reduction checker has to check those context conditions that are not yet checked by the grammar reader. Remember the context conditions:

(CC1) Each non-terminal must have at most one production.
(CC2) Each non-terminal must have at least one production.
(CC3) Each non-terminal must be productive.
(CC4) Each non-terminal must be reachable from the axiom.
(CC5) In list descriptions, the second non-terminal must equal the first one.

(CC1) and (CC5) are checked by the grammar reader where violation of (CC1) causes a fatal error and of (CC5) a warning. (CC2) through (CC4) are checked now.

If m+ was chosen, positive messages are issued if the conditions are satisfied. If a condition is violated, a list of violating non-terminal symbols is printed. Violation of (CC2) or (CC3) is a fatal error that lets the program exit, violation of (CC4) only results in a warning. Productions of unreachable non-terminals are simply cut off the grammar graph, then execution may be continued as normal. If condition (CC2) fails, (CC3) and (CC4) are not tested, if (CC3) fails, (CC4) is checked nevertheless.

3.3. The first- and follow-set computer

If option m+ was selected, this part starts by printing the message "First- and follow-set computation ...". Normally, it works silently without further messages, but the first-set computer is able to detect left recursions as side effect. It then prints "Non-terminal *N* is directly left-recursive" or "Left recursion detected. Involved non-terminals: *N1 N2* $\cdots$". *For each left recursive cycle, one such message is issued. Left recursion is indicated since it disallows the automatic solving of ELL(2) conflicts because of the danger to generate infinitely looping parsers.*

First- and follow-sets are computed without fixed point iteration by an efficient algorithm described in [3].

3.4. ELL condition checker

This part first looks for ELL(1) conflicts, then tries to solve them by ELL(2) computation, and if that fails, performs an automatic ELL(2) conflict solving without further look-ahead. If option e- was chosen, the ELL checker is not activated, with option e+, the automatic ELL(2) conflict solving is disabled. For the following description, we assume option e# such that all possible actions are really performed.

Since an efficient look-ahead-2 computation must be done globally, information about all found ELL(1) conflicts must be stored somewhere before ELL(2) computation is begun. If this storage is full, the remaining ELL(1) conflicts are forgotten and not solved. Then the message "There are too many (more than 1000) LL(1) conflicts" is issued and parser generation is disabled.

In rare cases, an overflow may occur during the look-ahead-2 computation itself. This is indicated by the message "Stopped. There are too many LL(1) conflicts under terminal *T*". The only remedy in all these cases is to simplify the input grammar and to try to reduce the number of ELL(1) conflicts.

The description of conflicts is controlled by the *p* option. If p- is chosen, only ELL(2) conflicts are described, otherwise ELL(1) conflicts, too. A conflict description starts by a header that looks like

> ELL(*k*) conflict in alternative expression in production of *N*
> (1) *(first alternative)*
> (2) | *(second alternative)*
> *etc.*

or

ELL(k) conflict in list description *LD* in production of N
between (1) entering and (2) skipping list

or

between (1) expansion and (2) end of list

Note that a minor disadvantage of this message is that the user cannot see which alternation is conflicting in productions like N: $(A|B)$ $(A|B)$. But we think that such productions are unlikely.

The list description *LD* and the alternatives are printed by a deparser working on the grammar graph. Thus they are not completely equal to the original regular subexpression in the input grammar, but re-formatted and unified with respect to semantically equivalent external representations, e.g. keywords are always printed upper case. More serious differences may result from the automatic removal of superfluous parentheses.

If p+ was selected, it follows a list of three or four kinds of look-ahead sets: first the complete look-ahead-1 sets of the various alternatives, second those subsets that cause the ELL(1) conflict. Then for each look-ahead-1 symbol that causes a conflict, the complete look-ahead-2 sets are listed, and at last, their subsets causing an LL(2) conflict (if there is such a conflict). With option p-, only the conflicting subsets of look-ahead-2 sets are listed. These listings of sets are preceeded by symbolic numbers as introduced in the description header to relate them to the various alternatives.

A special situation that one might consider as ambiguity and thus as LL(k) conflict for any k is already solved by the algorithm of conflict detection and thus not listed as ELL(1) conflict. It is the matter of list constructs $\{N \ldots\}$ without separating terminal where the repeated non-terminal N derives ε. Here the symbol ε is automatically removed from the first-set of N such that the generated parser will never reduce ε to N (if it did so in some contexts, it would do it for ever and thus fall into an infinite loop).

If an ELL(2) conflict occurs, option e# was selected, and there is no left recursion, the automatic conflict solving is activated. The solution of conflicts at list descriptions is easy: it is always decided to choose alternative (1) such that the generated parser will stay as long as possible in a list construct. Conflicts at alternations are treated more sophisticated: if the conflict arises under look-ahead-1 symbol T, the various alternatives involved in the ELL(2) conflict are divided into two classes: those containing T in their first-set, and the other ones (thus containing ε in their first-set and T in their follow-set). Then the generator decides that the generated parser always has to select the lexical first alternative of the first class, and if this class is empty, the lexical first one of the second class.

Take as example the first alternation in the production

N : (a c | a b | a |) (a | b) b.

There is an ELL(2) conflict under look-ahead string 'a b'. Involved alternatives are 'a b', 'a', and ' '. 'a b' and 'a' form the first class, and ' ' the second class, thus 'a b' is chosen.

If the production had been written

N : (| a | a b | a c) (a | b) b.

then ' ', 'a', and 'a b' are involved, too, the first class consisting of 'a' and 'a b' in this order, and the second class of ' '. Then 'a' is chosen. Naturally, the parser generated from this grammar will not accept the full language, e.g. 'a b' and 'a b a b' will not be accepted.

There is an important kind of ELL(2) conflict that is correctly solved without diminishing the language accepted by the generated parser. It is the matter of the well-known "dangling else" problem that makes many programming languages, including Pascal and C, failing to satisfy the LL property. The critical production looks like

If-statement : if Expression then Statement (| else Statement) .

The alternation is not LL(k) decidable for any k when the look-ahead-1 symbol is *else*. The automatic solver chooses the non-empty alternative no matter whether it is written down first or not. This is the correct solution, it means that each *else* part belongs to the closest previous *else*-less *if*.

3.5. Tree condition checker

For the following description, we shall call non-terminals, free operators, and list descriptions important regular factors. Free occurrences of operators are those that don't appear after *root* or in a list description. With these notions, we can state the first three tree conditions shorter than in chapter 2.7:

Assume the right hand side of each production has been transformed into an alternation of concatenations of regular factors. Then

(TC1) Each concatenation contains at most one root-statement.

(TC2) Each concatenation contains at least one important factor.

(TC3) If there is more than one important factor in a concatenation, there must be a root-statement in it.

The tree condition checker does not really transform the regular expressions, it uses an attribution with one integer-attribute for each condition as efficient simulation.

If t- was selected, no condition is checked, if t+, the three conditions above are tested, and with t#, all four conditions are checked. The fourth condition is the arity condition, it is considered below.

The existence of a correct string-to-tree parser depends on the satisfaction of the first three tree conditions. Thus their violation disallows generation if option *gt*, *gm*, or *gi* was selected. The arity condition is only obliged if an Optran tree grammar is to be produced (with option o+ or o#).

If a tree condition is violated, a header is printed (an example is given below), then it follows a list of violations:

Production of *N* *(erroneous concatenations)*

For each production, only the worst concatenations are printed. They are combined into one expression by use of alternations of the untransformed regular expression.

The output algorithm is to go through the expression and to print the worst alternatives of each encountered alternation. Parentheses of alternations appear in the output even if only one alternative of theirs is printed.

An example will show the resulting messages. Note that it does not contain any terminal or non-terminal on the right hand side of the production, thus it is naturally nonsense and violates the ELL(2) condition. On the other hand, the examples of the previous chapter did not contain operators and violated the tree conditions.

N: (root ′r1′ |) (′op1′ |) (root ′r2a′ | root ′r2b′ |) (root ′r3′ | ′op2′).

Output:

```
Tree condition 1 violated:
There is more than one root statement in some concatenations

  Production of "N"
    ( ROOT 'r1' ) ( 'op1' | EMPTY ) ( ROOT 'r2a' | ROOT 'r2b' )
    ( ROOT 'r3' )

Tree condition 2 violated:
Some concatenations imply empty subtrees

  Production of "N"
    ( ROOT 'r1' | EMPTY ) ( EMPTY ) ( ROOT 'r2a' | ROOT 'r2b' | EMPTY )
    ( ROOT 'r3' )

Tree condition 3 violated:
Root statements are missing

  Production of "N"
    ( EMPTY ) ( 'op1' ) ( EMPTY ) ( 'op2' )
```

Explanation:

There are $2 \cdot 2 \cdot 3 \cdot 2 = 24$ different concatenations, most of them violating a tree condition. For (TC1), there are 2 concatenations without *root*s, 8 with 1 *root*, 10 with 2 *root*s, and 4 with even 3 *root*s. For the message, the worst alternatives from each alternation are taken. It is the first alternative from the first alternation, the complete second alternation since both alternatives are equally bad (or good in this case), etc. The result is a compact representation of the 4 concatenations with 3 root-statements. The other two messages are built analogously. Note that the output is done by a deparser converting keywords to upper case and using *EMPTY* to denote empty alternatives.

The arity condition

Each occurrence of an operator in any concatenation is associated with an implicit arity. The arity of an occurrence after *root* is the number of important factors in the concatenation, the arity of an operator in a left or right associative list is 2, in a non-associative list it is $*$, and free occurrences are associated with arity 0. Arity $*$ means that this operator may have an arbitrary number of children in a generated tree.

The arity condition (TC4) then demands that all occurrences of one operator are associated with the same arity. Arity $*$ is considered as incompatible with all arities of a number *n*.

If r- was selected, the trees constructed by the generated parser do really contain list operators with arbitrary arity. If r+ was chosen, they are replaced by recursive constructs such that operators in non-associative lists get arity 2, and the implicitly constructed end operators get arity 0. But in the generator, list operators are associated with arity $*$ and thus incompatible with arity 2, no matter whether r+ or r- was chosen. The purpose is to achieve the following property:

If a grammar satisfies the arity condition with option r+, then it will also satisfy it with option r-.

The inverse implication is not true because of the implicitly constructed operators.

Examples:

If-statement: if Expression then Statement (| else Statement) root ′if-stat′.

If this expression were transformed into an alternation of concatenations, it would consist of two concatenations both containing an occurrence of *root ′if-stat′*. But one concatenation contains two important factors (Expression, Statement), the other one three (Expression, Statement, Statement). The error message will be:

```
Inconsistent use of operator ′if-stat′ in production of "If-statement"
  Arity may be  2 or  3 by root statement
```

This kind of message always contains two arities, namely the minimum and maximum arity implied by the different concatenations. The user could claim that this message be too laconic, but (s)he has anyway to analyze the erroneous production when correcting it.

A correct solution for this production would be

```
If-statement:        if Expression then Statement
                     (′no-else′ | else Statement)  root ′if-stat′.
```

or

```
If-statement:        if Expression then Statement
                     (root ′if-then′ | else Statement  root ′if-then-else′).
```

Another example: A: {′list′ N ...}.   B: C D root ′list′.
where A, B, C, D, N being non-terminals.
Produced message:

Inconsistent use of operator ′list′
  Arity $*$ in production of "A" within the construct
    { ′list′ N ... N }
  Arity 2 in production of "B" within the construct
    ROOT ′list′

If the list construct were declared as left or right associative, no error message would be printed. The error may be corrected by choosing different names for the two occurrences of *′list′*.

Assume that there are four productions where the same operator occurs with arity 2, 3, 2, and 3 in this order. Then three independent error messages (2↔3, 3↔2, 2↔3) will be produced since the arity checker remembers only the last occurrence of each operator with its arity.

An example for the difference between r- and r+:
  A: {′list′ N ...}.   B: C D root ′empty_list′.

With r-, there is no problem, with r+empty_, there is an inconsistence between the occurrence of *′empty_list′* constructed at the list description (arity 0), and the occurrence in the production of *B* (arity 2). If the production of *B* had preceeded the production of *A*, then the grammar would have been rejected already by the grammar reader due to technical reasons.

3.6. Optran tree grammar computation

If the generated parser shall be used as front-end for Optran, Optran must get a tree grammar describing the set of trees that will be submitted to Optran by the generated parser. This tree grammar is part of a so-called Optran t-unit and may be either written by the user or computed by the generator. Option *a* controls whether scanner attributes are attached to tree productions (a+) or not (a-). Scanner attributes are always declared as imported local attributes by our generator. They are attached to tree productions deriving an operator that has at least one occurrence with an associated terminal $[T]$ in the input grammar.

Optran tree grammars consist of a sequence of productions $X ::= Y$ or $X ::= < op, X_1, \cdots, X_k >$ where X, Y, X_i are non-terminals, and $'op'$ is an operator with arity k. Operators are written without quotes in Optran. They can be distinguished from non-terminals since they may only occur immediately after a left angular bracket '<'. Trees on right hand sides of productions are given in prefix notation, thus the example tree consists of a root marked by operator *op*, and *k* children.

Note that in Optran, there may exist many productions with the same left hand side. Thus a tree production in Optran corresponds to that what we have called concatenation and consider as part of a greater production.

The implicitly defined tree grammar *G*" of chapter 2.7 cannot be printed directly as Optran tree grammar since there are severe restrictions on Optran grammars.

(1) The forms of productions given above are the only legal forms. This means that trees on the right hand side of productions may have a depth of at most 1, and operators cannot appear as children in these trees.

(2) The second restriction is more serious: a given operator may occur only in exactly one production.

The second restriction implies that the real tree grammar must be modified by additional non-terminals - to achieve an approximation that is as good as possible - or by combining all non-terminals into one - to achieve an approximation that is as bad as possible. Even the best approximation does in general not describe the real tree language exactly, but a superset of it.

If option o# was chosen, the generator computes an Optran grammar that is as close as possible at the real grammar *G*". This may be very time and space expensive since the transformation of regular expressions of the input grammar into alternations of concatenations must be really performed - time needed for this may be exponentionally depend on the input grammar size - and the satisfaction of restriction (2) can only be guaranteed by global comparison of the resulting concatenations - requiring to store the complete transformed grammar somewhere. Thus a storage overflow may occur indicated by the message "Stopped. Production of *N* is too huge for tree grammar computation".

Because of the exponential time and space complexity, an option o+ is provided that causes a simple

Optran tree grammar to be computed in linear time and space. It is the worst grammar still respecting the names of operators and their arity. It results from combining all non-terminals into one (called S), and thus consists of a sequence of productions $S ::= < op, S, \cdots, S >$ - one for each operator $'op'$ - where the number of occurrences of S on the right hand side equals the arity of $'op'$.

Whereas this simple grammar is now completely described, we shall give some examples to illustrate how the grammar produced with option o# looks like. Non-terminals will be written capitalized, terminals completely lower case.

A: $(a \mid b)\ B \mid C$ leads to $A ::= B;\ A ::= C;$

Originally there are three concatenations and thus three tree productions. But two of them are equal because the terminals a and b are not important. The generator compares the arising tree productions such that no production is printed twice.

If_statement:	if Expression then Statement ('no_else' | else Statement) root 'if_stat'.

leads to

If_statement	::= <if_stat, Expression, Statement, if_stat_3>;
if_stat_3	::= <no_else>;
if_stat_3	::= Statement;

The generator must introduce an additional non-terminal *if_stat_3* as third child of operator *'if_stat'* since this child is not constant in the various concatenations with root *'if_stat'*. The solution

If_statement ::= <if_stat, Expression, Statement, no_else>;
If_statement ::= <if_stat, Expression, Statement, Statement>;

would be a violation of both restrictions (1) and (2).

Assume you have production

Constant:	(root 'sign' [addop] |) ('const_number' [number] | 'const_id' [id]).

in the type definition part, and production

Factor:	$\cdots$ | root 'sign' [addop] Factor | $\cdots$.

in the expression definition part of a specification of some Pascal-like programming language. Without the restrictions, the resulting tree productions would be

Constant	::= <sign, const_number>;
Constant	::= <sign, const_id>;
Constant	::= <const_number>;
Constant	::= <const_id>;
Factor	::= <sign, Factor>;
Factor	::= $\cdots$

The three productions containing *'sign'* must be unified to one. A non-terminal *sign_0* is constructed as left hand side of the unified production since the three original productions don't have the same left hand side. In addition, a non-terminal *sign_1* is needed since the three productions have different objects on their first child position. Then there are two productions deriving *'const_number'* with differing left hand sides, namely *Constant* and *sign_1*. Therefore, non-terminals *const_number_0* and analogously *const_id_0* are constructed. The result is

Constant	::= sign_0;
sign_0	::= < sign, sign_1 >;

```
sign_1            ::= const_number_0;
const_number_0    ::= < const_number >;
sign_1            ::= const_id_0;
const_id_0        ::= < const_id >;
sign_1            ::= Factor;
Constant          ::= const_number_0;
Constant          ::= const_id_0;

Factor            ::= sign_0;
Factor            ::= · · ·
```

Note that these productions are printed with option a- to simplify them. With a+, the second production would, for example, look like

```
sign_0      ::= < sign, sign_1 >;
   local scanattr: scanattrtyp;
   imported scanattr;
```

due to the terminal *addop* attached to the operator *′sign′*.

Associative list descriptions describe recursive trees:

```
A: {B ′binop′ [t] ...} left.      →   A ::= B;  A ::= <binop, A, B>;
A: {B ′binop′ [t] ...} right.     →   A ::= B;  A ::= <binop, B, A>;
```

Non-associative lists must be replaced by a recursive tree construct, since actual Optran does not allow for list operators with variable arity. An additional end operator is constructed implicitly by the generator. With standard prefix *empty_*, the resulting tree productions look like

```
A: {′list′ B ...}.                →   A ::= <empty_list>;  A ::= <list, A, B>;
```

Sometimes, an additional non-terminal is needed to "carry" the recursion:

```
A: {′list′ B ...} | C.
→      A          ::= list_0;
       list_0     ::= < empty_list >;
       list_0     ::= < list, list_0, B >;
       A          ::= C;
```

This is done to prevent *C* from being involved into the recursion.

The generated Optran tree grammar satisfies the two restrictions above and is normally correct, i.e. it is accepted by Optran and describes a superset of the set of trees built by the generated parser.

But there are some sources for errors that may cause the grammar to be incorrect. These errors are **not** indicated by our generator, thus the user him/herself is responsible to avoid them.

We shall list these possible errors for the case of option o#. Some of them are also possible if o+ is chosen.

(1) Names of non-terminals and operators in the input grammar are directly copied into the Optran tree grammar. Thus they must be Optran names, i.e. they may consist only of letters, digits, and underscores '_'. In addition, they must not equal Optran keywords. Terminal names of the input grammar don't occur in the Optran grammar, and thus they are not restricted to be Optran names.

(2) Names of non-terminals and operators are distinguished in the input grammar by quotes surrounding operator names. In Optran, these quotes are not written, thus it is possible that non-terminal and operator names become equal. The user should avoid this by observing a fixed lexical strategy to build non-terminal and operator names, e.g. capitalizing non-terminal names as in the examples above.

(3) If names of additional non-terminals are already used in the input grammar, an error will result that might be hard to detect. The user should therefore completely avoid names with suffix '_*n*', *n* being a number.

(4) If an operator name in the input grammar was rather long, and has eventually still grown up by a prefix like 'empty_', then the suffix '_*n*' of additional non-terminals might be partly beyond the 32th

position of the name. Then the number is not read by Optran and thus different non-terminals constructed from the same operator become equal. Thus the user should avoid operator names longer than, say, 24 characters.

3.7. The real code generator

If the code generator is activated, and m+ was selected, it issues the message "Parser generation ...". If code generation is disabled, either by option g- or by an error, the message "No parser generation" is printed.

The generated code is a fragment of a Pascal program and consists of a table of symbol names and their internal codes, information to treat "holes" in program fragments to be analyzed (only with *gm* or *gi*), a mapping associating a "right neighbour" to each parse state, and the parse table itself describing what actions are to be performed in a given parse state with given look-ahead. Drivers interpreting these informations are not generated, but included from constant files when the generated parser is compiled.

The parser table does not any longer look as described in [3] since many parse states are omitted and the remaining table is optimized such that the actions of as most states as possible are independent of the actual look-ahead. The actions specified in the parse table serve to consume input, to build up the abstract syntax tree, and to handle the parse stack.

The parse states correspond roughly to subexpressions of the right hand sides of productions. The right neighbour associated with each parse state corresponds to the next subexpression to the right in the actual production. A right neighbour being 0 means that the end of the actual production is reached and a "reduce" action is to be performed.

A symbolic listing of parse states, their right neighbour, and their corresponding regular subexpression can be got by the grammar graph printer.

3.8. The grammar graph printer

Output of the grammar graph is induced by option i+ or i#, first- and follow-sets are listed together with each node with option i#. This part of the generator is activated at the end of execution, thus grammar graph printing will not be performed if the input grammar was already rejected by input reader or reduction tester.

The grammar graph consists of a forest of trees, one tree for each production. These trees are abstract syntax trees for the regular expression on the right hand side. They are linked by additional edges between non-terminal leaves and roots of trees produced by the non-terminals.

The output of the graph starts by a summary of the input grammar (see example below). Then the trees are printed one by one preceeded by a list of "predecessors of the root". These are non-terminal leaves marked by the non-terminal on the left hand side of the production actually printed. The trees are listed in pre-order (parents before children) according to the format:

node number (parse state -> right neighbour) indent node description

node number:

The nodes in the grammar graph are numbered from 1 on. These numbers have no internal meaning and are produced only for the purpose of this listing.

(parse state -> right neighbour):

Some nodes (more exact: some subtrees) correspond directly to parse states in the generated parser. For these nodes, the number of the parse state and of its right neighbour state are given. At nodes without corresponding parse states, white space is printed between the parentheses. If code generation was disabled (by option g- or by errors), this information cannot be given since parse state numbers are computed during generation. If the input grammar failed to be ELL(1), there are look-ahead-2 states that do not correspond to nodes of the grammar graph, thus their numbers don't occur in this listing.

indent:

The indent consists of two strings. The first string consists of two blanks ' ' for nodes near the root of the tree. The second string consists of a sequence of substrings '| ' - one for each level of depth. If the second string becomes too large, it is reset to ε. After n resets where $1 \le n \le 9$, the first string is 'n>';

and after ten or more resets, it will be '>>'.

node description:

A readable form of the actual node and its contents

As example, assume you have the grammar:

```
Statement-list:   {'statement-list' Statement semicolon ...}.
Statement:        while Expression do Statement-list od  root 'while'
                  | 'variable' [id] assign Expression  root 'assignment'.
Expression:       {Atom 'binop' [binop] ...} left.
Atom:             'atom' [id] | 'atom' [constant] | left-par Expression right-par.
```

Listing of the grammar graph with option i+ after successful generation:

```
GRAMMAR GRAPH

Number of terminals:          10
Number of productions:         4
Number of nodes:              26
Number of operators:           6
Number of parse states:       22

Tree belonging to non-terminal "Statement-list"
  Predecessors of the root:    9

 1  ( 1 ->  0)         List description    Operator: 'statement-list'
                       Associativity: none
 2  ( 2 ->  3)         | Non-terminal "Statement"
 3  ( 3 ->  2)         | Terminal "semicolon"

Tree belonging to non-terminal "Statement"
  Predecessors of the root:    2

 4  ( 4 ->  0)         Alternation node with   2 successors
 5  (        )         | Product node with   6 successors
 6  ( 5 ->  6)         | | Terminal "while"
 7  ( 6 ->  7)         | | Non-terminal "Expression"
 8  ( 7 ->  8)         | | Terminal "do"
 9  ( 8 ->  9)         | | Non-terminal "Statement-list"
10  ( 9 -> 10)         | | Terminal "od"
11  (10 ->  0)         | | ROOT 'while'
12  (        )         | Product node with   4 successors
13  (11 -> 12)         | | Operator name 'variable' depending on terminal "id"
14  (12 -> 13)         | | Terminal "assign"
15  (13 -> 14)         | | Non-terminal "Expression"
16  (14 ->  0)         | | ROOT 'assignment'

Tree belonging to non-terminal "Expression"
  Predecessors of the root:    7  15  25

17  (15 ->  0)         List description    Operator: -    Associativity: left
18  (16 -> 17)         | Non-terminal "Atom"
19  (17 -> 16)         | Operator name 'binop' depending on terminal "binop"

Tree belonging to non-terminal "Atom"
  Predecessors of the root:   18
```

```
20  (18 ->  0)        Alternation node with   3 successors
21  (         )       | Operator name 'atom' depending on terminal "id"
22  (         )       | Operator name 'atom' depending on terminal "constant"
23  (         )       | Product node with   3 successors
24  (19 -> 20)        | | Terminal "left-par"
25  (20 -> 21)        | | Non-terminal "Expression"
26  (21 ->  0)        | | Terminal "right-par"
```

4. The generated parsers

4.1. Introduction

There are four different kinds of generated parsers. The *g* option of the generator controls which kind is generated:

gn → 'notr' parser
: This parser is able to analyze programs and to detect syntax errors. No trees are generated.

gt → 'tree' parser
: This parser reads complete programs of the specified language and translates them into abstract syntax trees according to the specifications of the generator input. It may also be used as front-end for Optran.

gm → 'muen' parser
: The multi-entry parser interactively translates program fragments and schemes into abstract syntax trees. It may be used to explore the properties of the specified string-to-tree translation such that the user can quickly check whether it is compatible with the one (s)he had in mind when writing the generator input.

gi → 'incr' parser
: The incremental string-to-tree parser interactively maintains a forest of abstract syntax trees corresponding to program fragments and schemes, and performs cut and paste operations on them.

Each kind of parser has its own parser driver (notrpas.i, treepas.i, etc.). The generated code that is combined with the constant driver during compilation also depends on the kind of parser it is generated for. Thus it is impossible to combine the same parse table with different drivers. Four complete runs of the generator on the same input (except the *g* option) are needed if all four kinds of parser shall be generated for one language. But the scanner generated by POCO does not depend on the value of the *g* option; thus the same scanner table may be included in all generated parsers. The only problem is that the name of the file containing the scanner table is always expected to be "*name*.scan.i" if the name of the generated parser is "*name*.p".

4.2. Common properties of the generated parsers

All parsers may be influenced by options similar to those of the generator. The two "batch" parsers 'notr' and 'tree' read options from the command line but not from input. The two interactive parsers 'muen' and 'incr' don't read the command line and provide standard values for options when started. They allow for the interactive changing of option values by menues.

If a syntax error is encountered, the parser outputs the line where the error is detected even if listing is disabled, then a line containing the character '^' pointing to the error position followed by the message "Error: *list of terminals* expected". There is no error recovery, and the parser stops reading at the error position.

When a parser is reading from a terminal, the end of input must be indicated by hitting the Unix-EOF-character ^D (control-D) at the beginning of a new line without following 'return' or 'line-feed'. If a syntax error was found, the parser stops reading immediately, thus ^D need not be hit in this situation (but an accidental hit is tolerated by the parser).

Abstract syntax trees are listed in readable form if the user wants this. The nodes are listed in pre-order, i.e. parents before children. For each node, the operator labeling it is printed, followed eventually by a scanner attribute. This is an instance of a terminal *T* bound to this operator by the construct $'op'$ $[T]$ in the generator input. First the terminal symbol class of the instance is printed. If the instance

belongs to an infinite terminal symbol class, the scanner generated by POCO is able to decode the internal code of the instance into its external form, and this form is printed surrounded by double quotes ’"’. Unfortunately, instances of fixed finite symbol classes cannot be decoded by the POCO generated scanner, then the parser prints the pair class code - relative code enclosed by parentheses, and the user must do the decoding him/herself.
The various kinds of terminal symbol classes and the internal codes of instances of terminal symbols are explained in chapter 2.4.

The syntax trees are listed with indent consisting of two strings, the second string being a sequence of ’| ’ substrings, one for each level of depth. If it has become too large, it is reset to ε. The first string is ’ ’ without resets, ’> ’ after one reset, ’>>’ after two or more resets.

It is possible to force the syntax tree to be printed even after a syntax error. The tree is then incomplete, the error position corresponds to the right most leaf of the tree fragment. Some nodes on the path from this leaf to the root may be already created, but not yet marked by an operator; they are listed with the pseudo operator "NOT YET DEFINED".

4.3. The ’notr’ parser

This kind of parser reads options from command line, a complete program to be analyzed from input, and prints eventual syntax error messages and on termination either "Accepted" or "Not accepted". Possible options are

l-	: no listing of input
l+	: listing enabled
m-	: message "Accepted" is suppressed
m+	: it is printed
fm*number*	: maximum line length for syntax error messages

Defaults are l+, m+, fm80. Called with l- and m-, the parser will output nothing if the parsed program is correct. The messages due to option errors are analogous to those of the generator (see chapter 2.2), but they are not introduced by a pointer line since the command line is not listed. An additional message is "Error: Option "*x*" is given twice".

4.4. The ’tree’ parser

This kind of parser also reads options from command line and a complete program from input. The program is translated into an abstract syntax tree that may be listed in readable form on output or in coded form on a file to be read by Optran.

Optran needs two input files if it shall work on an abstract syntax tree. Both files may have arbitrary names, but Optran assumes the names "decfile" and "infile" by default. Thus these names are also default names for the output of our parser.

The "decfile" contains a list of operator names and of instances of terminal symbols encountered in the translated program together with their internal codes. Optran expects the list to be in a special order that looks strange for human readers, but shall make the file easier to read for Optran since the order fits into the data structure Optran is using internally. A procedure computing this special order was provided by B. Weisgerber - a member of the Optran group - and is included in our parser during compilation. The "infile" contains the abstract syntax tree itself and is written in Optran code. It is not a textfile and thus neither directly readable for users nor safely portable to other computers.

Options for the ’tree’ parser are:

l-	: no listing of input
l+	: input is listed
Default	: l+
m-	: no output of readable syntax tree
m+	: output of readable tree iff program is correct
m#	: tree is printed even if the program was erroneous
Default	: m+
fm*number*	: Maximum line length for error messages and tree listing

Default	: fm80
o-	: "infile" is not created
o+	: "infile" is created
o+*name*	: "infile" is created and called *name*
d-	: "decfile" is not created
d+	: "decfile" is created
d+*name*	: "decfile" is created and called *name*
Defaults	: if neither *o* nor *d* is set explicitly, they are both set to '-', if one of them is defined to be '+', the other one is also set to '+'

If the parser was generated with option r-, non-associative list constructs are translated into flat "iterative" trees. If the generation was done with r+, the parser is able to produce likewise "iterative" and "recursive" trees from those list constructs. Then there is an additional option:

r- : produce "iterative" trees
r+ : produce "recursive" trees
Default is then r+.

Summarized, this means: generation with r- implies parser working with r-, and r+ is impossible; generation with r+ implies parser works by default with r+, but is also able to work with r-.

Note that actual Optran does not accept "iterative" trees, thus d+/o+ and r- are incompatible with each other. Therefore a parser generated with r- can never be used as front-end for Optran.

4.5. The 'muen' parser

The 'multi-entry' parser is able to read program fragments and schemes, and to translate them into abstract syntax trees. Thus it is a tool for interactive exploring the string-to-tree mapping specified in the generator input.

The 'base state' of the parser is a cycle starting with the message "Please give type name (Empty input means *previous type name*, ˆD -> menue)". The user may then enter a type name, that is essentially a non-terminal name, in the lexical form specified in the *typename* section of the generator input. See chapter 2.5 for the relation between non-terminal names and type names. If the user gives an empty input, i.e. only hits the 'return' or 'line-feed' key, the parser takes the type name used in the previous repetition of the cycle. If the parser has just begun to work, this previous type name is always the axiom of the input grammar.

When a type name has been chosen by the user, the parser demands for a construct derived by this type, printing the message "Please give *chosen type name*". The user may then enter a program fragment or scheme of the selected type. This input may consist of many input lines and is normally ended by hitting ˆD at the beginning of a new line. It may contain "holes", i.e. subconstructs left unspecified. These holes must be typed by an appropriate type name.

After having analyzed the program fragment, the parser lists a readable form of the corresponding abstract syntax tree, then a new repetition of the base cycle starts.

If the user hits ˆD instead of entering a type name, the base cycle is left and a menue state is entered where values of options may be changed interactively. The parser does not read options from command line, and assumes default values at the beginning of its work. When the menue is printed the first time, it looks like (if the parser was generated with option r+):

(p)		Print previous tree again, then return to base state
(r)	List nodes:	recursive -> iterative
(f)	Maximum line length:	80 -> new value
(l)	Listing of input:	no -> yes
(m)	Trees are shown after syntax errors:	no -> yes
(o)	All settings okay; return to base state	
(@)	Exit program	

If 'r', 'l', 'm' is entered, the values around the arrow '->' are interchanged. The value before the arrow means the actual value, the one after the arrow means the value becoming valid if this row is chosen by the user. Listing of input lines is turned off by default since it looks ugly when the input is coming from a terminal. If 'f' is chosen, the parser aks for the new maximum line length by the message "Please give new value". Choosing 'r', 'l', 'm', or 'f' does not terminate the menue state, the menue is printed again. The menue state may be left by 'p' where first the actual tree is printed again, by 'o' where the base state is entered immediately, or by '@' where the parser program is terminated.

Note that the actual tree is not affected by changing the value of the *r* option. If you want to compare "recursive" and "iterative" tree of the same program fragment, you have to enter this fragment twice.

If the parser was generated with option r-, it is only able to construct "iterative" trees. The menue then looks like

(p)	Print previous tree again, then return to base state	
---	List nodes:	iterative (no change possible)
(f)	Maximum line length:	80 -> new value
(l)	Listing of input:	no -> yes
(m)	Trees are shown after syntax errors:	no -> yes
(o)	All settings okay; return to base state	
(@)	Exit program	

4.6. The 'incr' parser

The incremental parser shall maintain a forest of abstract syntax trees and perform cut and paste operations on them. For the time being, cut and paste are not yet implemented, thus this kind of parser is still a less comfortable version of the 'muen' parser. Furthermore, it is more likely to work erroneously since it is not yet tested thoroughly.

Appendix A: Summary of the syntax of the generator input

We shall use a regular right part grammar with the following meta-symbols:

::= "produces"
| alternative
***** Kleene-star: zero or more repetitions
+ one or more repetitions
ε empty string
( ) meta-parentheses for grouping
' character to improve readability (see below)

If two alternatives are semantically equivalent, we shall write **or** instead of **|**. Non-terminals and terminals of the meta-grammar will be written in italics, the non-terminals capitalized. Keywords and characters directly occurring in the input are written in Roman font as they are except blanks written as *BL*, tabs written as *TA*, and newline characters written as *NL*. Sometimes, characters are enclosed by round quotes ' to distinguish them from meta-symbols, e.g. '|', '(', ')'.

character means any arbitrary character,
visible-char means any character except *BL*, *TA*, and *NL*,
char-except-quote means any character except quote ′.

Tokens of the generator input

Some of these tokens are used in the real meta-grammar below, they are marked by an arrow. Other tokens are only used as parts of other tokens.

letter **::=** a | b | c etc. | A | B | C etc.
digit **::=** 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
usable-char **::=** *letter* | *digit* | ! | @ | # | $ | % | ^ | &
| _ | - | + | = | ~ | ‘ | \ | ; | " | < | > | , | ?
→ *POCO-char* **::=** ′ *char-except-quote* ′ | ′′′′
→ *string* **::=** (′ *char-except-quote* * ′) **+**

→ *word* ::= *visible-char* $^+$
→ *number* ::= *digit* $^+$
→ *option* ::= l- | l+ | m- | m+ | i- | i+ | i#
| e- | e+ | e# | p- | p+ | t- | t+ | t#
| g- | gn | gt | gm | gi | c:*word*
| s- | s+ | s# | s+*word* | s#*word*
| o- | o+ | o# | o+*word* | o#*word*
| a- | a+ | r- | r+ | r+*word*
| fm*number* | fg*number* | fo*number*
→ *POCO-name* ::= *letter* (*letter* | *digit* | _) *
symbol-name ::= *usable-char* $^+$
→ *terminal* ::= *symbol-name*
→ *non-terminal* ::= *symbol-name*
→ *operator* ::= ′ *usable-char* $^+$ ′

Ignorable tokens:

space ::= (*BL* | *TA* | *NL*) $^+$
comment ::= '(∗' *character* * '∗)'
The sequence of characters must not contain '∗)'

Meta-grammar for the input:

The meta-grammar defines sequences of tokens that may form the generator input. Ignorable tokens may be interspersed everywhere among these tokens. The *word* at the beginning of the input is the name of the called generator program that is always ignored. It is automatically part of the input since the command line is considered as first line of input.

Input ::= *word*
Options-section Classes-section Terminals-section
Typename-section Axiom-section Productions-section

Options-section ::= *option* *

Classes-section ::= ε | (CLASSES **or** classes) *Char-class-definition* *
Char-class-definition ::= (- | ε) *POCO-name* = *Char-class-def* ;
Char-class-def ::= *Char-class* (, *Char-class*) *
Char-class ::= *POCO-char* | *POCO-char* - *POCO-char*

Terminals-section ::= ε | (TERMINALS **or** terminals)
Symbol-class-definition *
Symbol-class-definition ::= *Symbol-class-mode terminal Lexical-specification*
Symbol-class-mode ::= ε | + | ∗ | -
Lexical-specification ::= ε | ˜ | = *POCO-expression* ;

POCO-expression ::= *POCO-factor* $^+$
POCO-factor ::= '!' *terminal* | *POCO-name* | *POCO-char* | *string*
| *Alternation* | *Equivalence*
| *Iteration* | *Allbut-expression*
Alternation ::= '(' *POCO-Expression* ('|' *POCO-expression*) * ')'
Equivalence ::= '(' *POCO-Expression* (, *POCO-expression*) * ')'
Iteration ::= [*POCO-expression*] | ∗ *Range* [*POCO-expression*]

Range ::= *number* - *number* | *number* - | - *number* | ε

Allbut-expression ::= allbut '(' *POCO-expression* ')'

Most of the productions above are taken from the POCO-manual [1] and only modified by minor changes and corrections.

Typename-section ::= ε | (TYPENAME **or** typename) terminal
(ε | (FIRSTCHAR **or** firstchar) *visible-char*)
(ε | (LASTCHAR **or** lastchar) *visible-char*)

Axiom-section ::= ε | (AXIOM **or** axiom) *non-terminal*

Productions-section ::= ε | (PRODUCTIONS **or** productions)
Production * (FINIS **or** finis)

Production ::= *non-terminal* : *Regular-expression* .

Regular-expression ::= *Regular-term* (('|' **or** '/') *Regular-term*) *

Regular-factor ::= '(' *Regular-expression* ')'
| EMPTY **or** empty | *non-terminal*
| *terminal* | *Node-description*
| (ROOT **or** root) *Node-description*
| *List-description*

Node-description ::= *operator* | *operator* [*terminal*]

List-description ::= { *operator* }
| { (*operator* | ε) *non-terminal* (*terminal* | ε)
(.. **or** ... | ..0.. **or** ...0...) (*non-terminal* **or** ε) }
| { *non-terminal* *operator* ([*terminal*] | ε)
(.. **or** ...) (*non-terminal* **or** ε) } *Assoc-spec*

Assoc-spec ::= LEFT **or** left | RIGHT **or** right

References

[1] M. J. Eulenstein: POCO - Compiler Generator User Manual
Technical Report Nr. A 2/85, Universit::=at des Saarlandes, Saarbr::=ucken, 1985

[2] M. Greim, S. Pistorius, M. Solsbacher, B. Weisgerber:
POPSY and OPTRAN Manual
PROSPECTRA Deliverable Item S.1.6-R-3.0, 1986

[3] R. Heckmann: An Efficient ELL(1) Parser Generator
Acta Informatica 23, 127 - 148, 1986

Table of contents