Lark - An LALR(2) Parser Generator With Backtracking

J. Grosch

DR. JOSEF GROSCH

COCOLAB - DATENVERARBEITUNG

GERMANY

Cocktail

Toolbox for Compiler Construction

Lark - An LALR(2) Parser Generator With Backtracking

Josef Grosch

July 15, 2005

Document No. 32

Dr. Josef Grosch
CoCoLab - Datenverarbeitung
Breslauer Str. 64c
76139 Karlsruhe
Germany

Phone: +49-721-91537544
Fax: +49-721-91537543
Email: grosch@cocolab.com

1. Introduction

Lark is a parser generator for LALR(2) and LR(1) grammars. With its backtracking facility it is even able to generate parsers for non-LR(k) languages. It is compatible with its predecessor, the parser generator *Lalr* [GrV]. The parser generator *Lark* offers the following features:

- generates highly efficient parsers
- provides automatic error reporting, error recovery, and error repair
- generates quickly detailed information about LR conflicts
- processes LALR(2) and LR(1) grammars
- supports backtracking for parsing non-LR(k) languages
- offers semantic predicates for the control of parsing by conditions
- supports named attributes as well as the $n notation
- provides a trace of parsing steps during run-time
- handles the bnf operator | which makes preprocessing with bnf rarely necessary

The parser generator *Lark* was originally developed with the aim to overcome the restrictions of LALR(1) grammars by the construction of an LR(k) parser generator. The name of the program is derived from this original goal. However, like others, we experienced the exponential complexity of LR(k) parser generation. Therefore we abandoned the original goal and adopted a method of user-controlled backtracking inspired by [Mer93]. This method overcomes even the restriction of LR(k) grammars of having a bounded lookahead of at most k tokens because it allows unbounded lookahead. Of course, backtracking is not for free. It slows down parsing but only in those parts of a grammar where backtracking is used. Without backtracking, the parsers generated by *Lark* are as highly efficient as parsers generated by *Lalr* [Gro88]. The reason is that *Lark* uses an extended version of the approved parser skeleton of *Lalr* [Gro90]. The method for the computation of the lookahead sets is a combination of the algorithms published in [KrM81, PCC85].

In this paragraph we describe further features of the parser generator. Every grammar rule may be associated with a semantic action consisting of arbitrary statements written in the target language. Whenever a grammar rule is recognized by the generated parser, the associated semantic action is executed. A mechanism for S-attribution (synthesized attributes) is provided to allow communication between the semantic actions. In case of LR conflicts, derivation trees are printed that ease the location of the problem. The conflict can be resolved by specifying precedence and associativity for terminals and rules or by the use of predicates. There are syntactic predicates and semantic predicates which are only used in case of LR conflicts. A rule with a semantic predicate is recognized only if the predicate yields true during run time. A rule with a syntactic predicate is recognized only if the lookahead tokens conform to the predicate. The most comfortable form of a syntactic predicate is backtracking where an arbitrary number of lookahead tokens is parsed in order to check whether it can be derived from a given nonterminal. Syntactic errors are handled fully automatic by the generated parsers including error reporting, error recovery, and error repair. The generated parsers are table-driven. The so-called comb-vector technique is used to compress the parse tables. The parse stack is implemented as a flexible array in order to avoid overflows. Parsers can be generated in the languages C, C++, Java, Modula-2, Ada, and Eiffel. Parsers generated by *Lark* are two to three times faster than *Yacc* [Joh75] generated ones. They reach a speed of two million lines per minute on a SPARC station ELC excluding the time for scanning. The size of the parsers is slightly increased in comparison to *Yacc* because of the comfortable error recovery and because *Lark* prefers to minimize run time rather than program size.

Besides the input language described in this manual, there is a second possibility which can be used to describe grammars for *Lark*. This alternate possibility is described in the document entitled "Preprocessors" [Groa]. The use of the language defined in the current manual works perfectly. However, in comparison to the alternate method, it is relatively low level. Therefore we recommend to use the language described in "Preprocessors". It offers the following advantages:

- The syntax is the same as for the tools *ast* [Grob] and *ag* [Groc] of the Cocktail Toolbox for Compiler Construction [GrE90].
- It allows for the automatic derivation of most of a scanner specification from a parser specification.
- The coding of tokens is done automatically and kept consistent with the scanner specification.
- The S-attribution or attribute grammar is checked for completeness and whether it obeys the SAG property.
- Declarations for the type *tParsAttribute* and the procedure *ErrorAttribute* are derived automatically from the attribute declarations.
- The grammar and the semantic actions might be separated into several modules.
- One disadvantage has to be mentioned: Predicates and backtracking are not yet supported by the alternative input language.

The rest of this manual is organized as follows: Section 2 explains the input language of the parser generator that is used to describe grammars. Section 3 discusses the classes of LALR(1) and LR(1) grammars. Section 4 deals with LR conflicts and ambiguous grammars. Section 5 explains the information generated about LR conflicts. Section 6 describes the optional listings of terminals, nonterminals, and grammar rules as well as the readable output of the generated parsing automaton. Section 7 describes the interfaces of the generated parsers. Section 8 discusses the error recovery of the generated parsers. Section 9 describes the trace of parsing actions that can be requested from a generated parser. Section 10 contains the manual page of the parser generator that summarizes all possible options. Appendix 1 summarizes the syntax of the input language. The Appendices 2 to 5 present examples of parser specifications.

2. Language Description

A parser generator takes as input a language description and it produces as output a parser. A parser is a procedure or a program module for analyzing a given input according to the language description. A language is described conveniently by a context-free grammar. A complete language description suitable as input for *Lark* is divided into the following parts which can be given in any order except for the grammar part which has to be last:

names for scanner and parser modules
target code sections
specification of the tokens
specification of precedence and associativity for operators
specification of start symbols
specification of the grammar

The part specifying the grammar is mandatory - all other parts are optional. The following sections discuss these parts as well as the lexical conventions. Appendix 1 summarizes the syntax of the input language using a grammar.

2.1. Lexical Conventions

A language description for the parser generator *Lark* can be written in free format. It consists primarily of identifiers, keywords, numbers, strings, delimiters, target-code, and comments.

An identifier is a sequence of letters, digits, and underscore characters '_'. The sequence must start with a letter or an underscore character '_'. Upper and lower case letters are distinguished. An identifier may be preceded by a backslash character '\' e. g. in case of conflicts with keywords. Such a construct is treated as an identifier whose name consists of the characters without the backslash character. Identifiers denote terminal and nonterminal symbols.

```
Factor   Term_2   \BEGIN
```

The following keywords are reserved and may not be used for identifiers:

```
BEGIN          CLOSE          EXPORT         GLOBAL         LEFT
LOCAL          NONE           OPER           PARSER         PREC
RIGHT          RULE           SCANNER        START          TOKEN
```

A number is a sequence of digits. Numbers are used to encode the tokens. The number zero '0' is reserved as code for the end-of-file token.

```
1   27
```

A string is a sequence of characters enclosed either in single quotes "'" or double quotes '"'. If the delimiting quote character is to be included within the string it has to be written twice. Strings denote terminal symbols or tokens. We will use the terms token and terminal as synonyms in this manual.

```
':='   "'"   ''''   "BEGIN"
```

The following special characters are used as delimiters:

```
=    :    .    |    ?    -    [    ]    {    }
```

So-called target-code actions or semantic actions are arbitrary declarations or statements written in the target language and enclosed either in curly brackets '{' and '}' or in edged brackets '[' and ']'. The characters '{' and '}' or '[' and ']' can be used within the actions as long as they are either properly nested or contained in strings or in character constants. Otherwise they have to be escaped by a backslash character '\'. The escape character '\' has to be escaped by itself if it is used outside of strings or character constants: '\\'. In general, a backslash character '\' can be used to escape any character outside of strings or character constants. The escape conventions are disabled within those tokens and the tokens are accepted unchanged. The actions are copied more or less unchecked and unchanged to the generated output. Syntactic errors are detected during compilation.

```
{ int x; }
{ printf ("}\n"); }
[ printf ("named and $n-attributes: %d %d", $name, $2); ]
```

The parser generator does not know about the syntax of the target language except for strings. Strings are checked for correct syntax in statements as well as in comments, because the tool does not distinguish between between statements and comments. This has the advantage that strings are copied unchanged to the generated output. However, it has the disadvantage that single and double

quotes have to appear in pairs in comment lines contained in semantic actions. Unpaired quotes are reported as incorrect strings.

```
{ printf ("hello \" and '\n"); /* it''s time to "go home" */ } (* correct   *)
{ printf ("hello \" and '\n"); /* it's time to "go
                                  home"                  */ } (* erroneous *)
```

There are two kinds of comments: First, a sequence of arbitrary characters can be enclosed in '(*' and '*)'. This kind of comment can be nested to arbitrary depth. Second, a sequence of arbitrary characters can be enclosed in '/*' and '*/'. This kind of comment may not be nested. Both kinds of comments may be used anywhere between lexical elements.

```
(* first  kind of comment *)
(* a (* nested *) comment *)
/* second kind of comment */
```

2.2. Names for Scanner and Parser

A grammar may optionally be headed by names for the files and modules to be generated:

```
SCANNER Identifier PARSER Identifier
```

The first identifier specifies the module name of the scanner to be used by the parser. The second identifier specifies a name which is used to derive the names of the parsing module, the parsing routine, etc. If the names are missing they default to *Scanner* and *Parser*. In the following we refer to these names by <Scanner> and <Parser>.

If the target language is Java, these names may include a package name. The scanner may be in a different package to the parser as in this example:

```
SCANNER mydomain.scanners.JavaScanner
PARSER  mydomain.parsers.JavaParser
```

Here the parser name is *JavaScanner* and the generated class will include a package declaration placing it in *mydomain.parsers*. The scanner is always referred to by its fully qualified name so there is no need to add an import statement.

2.3. Target Code Sections

A grammar may contain several sections containing *target code*. Target code is code written in the target language. It is copied unchecked and unchanged to certain places in the generated module. Every section is introduced by a distinct keyword. The meaning of the different sections is as follows:

IMPORT:	declaration of external modules used by the generated parser.
EXPORT:	declarations visible to users of the generated parser.
GLOBAL:	declarations to be included in the implementation part or body at global level.
LOCAL:	declarations to be included in the parsing procedure.
BEGIN:	statements to initialize the declared data structures.
CLOSE:	statements to finalize the declared data structures.

The exact details vary according to the conventions of the target language, and will be discussed later.

Example in C or C++:

```
EXPORT { typedef int MyType; extern MyType Sum; }
GLOBAL {# include "Idents.h"
          MyType Sum; }
BEGIN  { Sum = 0; }
CLOSE  { printf ("%d", Sum); }
```

Example in Modula-2:

```
EXPORT { TYPE MyType = INTEGER; VAR Sum: MyType; }
GLOBAL { FROM Idents IMPORT tIdent; }
BEGIN  { Sum := 0; }
CLOSE  { WriteI (Sum, 0); }
```

Example in Ada:

```
IMPORT { with Idents; use Idents; }
GLOBAL { type MyType is Integer; Sum: MyType; }
BEGIN  { Sum := 0; }
CLOSE  { Put (Sum); }
```

Example in Eiffel:

```
EXPORT { Sum, }
GLOBAL { Sum: INTEGER; }
BEGIN  { Sum := 0; }
CLOSE  { io.output.putint (Sum); }
```

Example in Java:

```
EXPORT { public int sum; }
BEGIN  { sum := 0; }
CLOSE  { System.out.println (Sum); }
```

If a GLOBAL section is not specified then it defaults to the following which provides a minimal definition for the type tParsAttribute:

C	`typedef struct { tScanAttribute Scan; } tParsAttribute;`
Modula2	`TYPE tParsAttribute = RECORD Scan: Scanner.tScanAttribute; END;`
Ada	`type tParsAttribute is record Scan: Scanner.tScanAttribute; end record;`
Java	`# define yyParsAttribute java.lang.Object`

If an EXPORT section is not specified then it defaults to the following:

C++	`typedef struct { tScanAttribute Scan; } <Parser>_tParsAttribute;`

2.4. Specification of Terminals

The terminals or tokens of a grammar may be declared by listing them after the keyword TOKEN. The tokens can be denoted by strings or identifiers. Optionally, an integer can be given to be used as

internal representation. Missing codes are added automatically by taking the lowest unused integers. The codes must be greater than zero. The code zero ‘0‘ is reserved for the end-of-file token. Undeclared tokens are allowed. They are declared implicitly and reported by a warning message. A token is undeclared if it is not declared as token and it is either denoted by a string or by an identifier which is not used as a nonterminal.

Example:

```
TOKEN
   "+"     = 4
   ':='
   ident   = 1
   'BEGIN'
   END     = 3
```

The token ':=' will be coded by 2 and 'BEGIN' by 5.

The declaration of a token can optionally be followed by a cost value and an external representation. This information is used by the error recovery. If tokens are to be inserted during error repair then tokens are selected according to minimal costs. Cost values have to be greater than zero. Missing cost values default to 10. An external representation is specified by a string. This string will be used for error messages. If no external representation is specified then the identifier or the string denoting the token will be used. Appendix 1 contains the exact syntax for the declaration of tokens.

Example:

```
TOKEN
   identifier = 1, 20, "name"                /* cost = 20 */
   number     = 2,  5, '0'                   /* cost =  5 */
   dummy        3, ""                        /* cost = 10 */
```

2.5. Precedence and Associativity for Operators

Sometimes grammars are ambiguous and then it is not possible to generate a parser without additional information. Ambiguous grammars can be turned into unambiguous ones in many cases by the additional specification of precedence and associativity for operators. Operators are tokens used in expressions. The keyword PREC may be followed by groups of tokens. Every group has to be introduced by one of the keywords LEFT, RIGHT, or NONE. The groups express increasing levels of precedence. LEFT, RIGHT, and NONE express left associativity, right associativity, and no associativity.

Example:

```
PREC
   NONE  '='
   LEFT  '+' '-'
   LEFT  '*' '/'
   RIGHT '**'
   LEFT  UNARY_MINUS
```

The precedence and associativity of operators is propagated to grammar rules or right-hand sides. A right-hand side receives the precedence and associativity of its right-most operator, if it exists. A right-hand side can be given the explicit precedence and associativity of an operator by adding a so-called PREC clause. This is of interest if there is either no operator in the right-hand side or in order

to overwrite the implicit precedence and associativity of an operator.

Example:

```
expression : '-' expression PREC UNARY_MINUS . /* overwrite binary '-' */
expression : expression expression PREC '+'  . /* no operator in rule  */
```

2.6. Start Symbols

One or more nonterminal symbols can be defined to act as so-called *start symbols*. The names of these nonterminals are listed after the keyword START. If this START clause is missing then the nonterminal on the left-hand side of the first grammar rule is implicitly defined as start symbol. The generated parser checks its input with respect to a certain start symbol. It checks whether the input can be derived from the start symbol given as argument.

Example:

```
START program statement expression
```

2.7. Grammar Rules

The core of a language description is a context-free grammar. A grammar consists of a set of rules. Every rule defines the possible structure of a language construct such as statement or expression. The following example specifies a trivial programming language.

Example:

```
RULE

statement  : 'WHILE' expression 'DO' statement ';'
           | 'IF' expression 'THEN' statement ';'
           | identifier ':=' expression ';'
           .
expression : term
           | expression '+' term
           .
term       : factor
           | term '*' factor
           .
factor     : number
           | identifier
           | '(' expression ')'
           .
```

A grammar rule is introduced by a left-hand side and a colon ':'. It is terminated by a dot '.'. The left-hand side has to be a nonterminal which is defined by the rule. Nonterminals are denoted by identifiers. An arbitrary number of rules with the same left-hand side may be specified. The order of the rules has no meaning except in the case of LR conflicts (see section 4). If no START clause is specified then the nonterminal on the left-hand side of the first rule serves as start symbol of the grammar.

For the definition of nonterminals we use nonterminals itself as well as terminals. Terminals are the basic symbols of a language. They constitute the input of the parser to be generated. Terminals are denoted either by identifiers or strings. A rule can specify several right-hand sides separated by a bar character '|'. The nonterminal *statement* from the above example could alternatively

be defined in the following style with the same meaning:

Example:

```
RULE

statement  : 'WHILE' expression 'DO' statement ';' .
statement  : 'IF' expression 'THEN' statement ';' .
statement  : identifier ':=' expression ';' .
```

2.8. Semantic Actions

Semantic actions serve to perform syntax-directed translation. This allows for example the generation of an intermediate representation such as a syntax tree or of a sequential intermediate language. A semantic action is an arbitrary sequence of statements written in the implementation language which is enclosed in curly brackets '{' and '}'. One or more semantic actions may be inserted in the right-hand side of a grammar rule. This first form of a semantic action is more precisely called a *conditional* semantic action. It is termed conditional because by default it is not executed when the parser does backtracking (trial mode). There is a second form of semantic actions which are enclosed in edged brackets '[' and ']'. This form is called *unconditional* because these semantic actions are executed by default during backtracking (trial mode) as well as during regular parsing (standard mode). The execution of semantic actions can be additionally controlled by an action flag and so-called selection masks. The section about "Reparsing" describes the details. The generated parser analyzes its input from left to right according to the specified rules. Whenever a semantic action is encountered in a rule its statements are executed.

The following grammar completely specifies the translation of simple arithmetic expressions into a postfix form for a stack machine.

```
RULE

expression : term
           | expression '+' term   { printf ("ADD\n"); }
           | expression '-' term   { printf ("SUB\n"); }
           .
term       : factor
           | term '*' factor       { printf ("MUL\n"); }
           | term '/' factor       { printf ("DIV\n"); }
           .
factor     : 'X'                   { printf ("LOAD X\n"); }
           | 'Y'                   { printf ("LOAD Y\n"); }
           | 'Z'                   { printf ("LOAD Z\n"); }
           | '(' expression ')'
           .
```

A parser generated from the above specification would translate the expression `X * ( Y + Z )` to

```
LOAD X
LOAD Y
LOAD Z
ADD
MUL
```

Due to the parsing method, semantic actions can only be executed when a complete rule has been recognized. This would imply that semantic actions have to be placed at the end of rules, only.

This location for semantic actions is the recommended one. Semantic actions within the right-hand side or even at the beginning of the right-hand side are possible. The grammar is transformed internally in this case by moving all semantic actions to the end of right-hand sides. This is done by the introduction of new nonterminals and new rules with empty right-hand sides.

Example:

```
The rule          X : u { A; } v .
is turned into    X : u Y v .
and               Y : { A; } .
```

Y is a new nonterminal different from all existing nonterminals. In rare cases, a grammar may lose its LALR(1) or LR(1) property due to the above transformation:

Example:

```
X : u v | u { A; } v w .
```

Without the semantic action { A; } this rule is LALR(1). With the semantic action and after the above transformation it is not LALR(1) any more. In such a case, the rules for conflict resolution may still lead to a working parser (see section 4).

2.9. Attributes: Definition and Computation

The parsers generated by *Lark* include a mechanism for a so-called S-attribution. This allows the evaluation of synthesized attributes during parsing. Attributes are values associated with the nonterminal and terminal symbols. The attributes allow for the communication of information among grammar rules and from the scanner to the parser. Attribute values are computed within semantic actions.

Attribute storage areas are maintained for all occurrences of grammar symbols. These storage areas are of the type *tParsAttribute*. This type has to be defined by the user in the GLOBAL target code section. Usually this type is a union or variant record with one member or variant for every symbol that has attributes. Every member or variant may be described by a struct or record type if a symbol has several attributes. There must always be a member called *Scan* of type *tScanAttribute*. The latter type is exported by the scanner. During the recognition of terminals this member is automatically supplied with the information of the external variable *Attribute* that is exported by the scanner, too. This variable provides additional data (the attributes) of terminals.

Example in C or C++:

```
typedef union {
   tScanAttribute Scan;
   tTree          Statement;
   tValue         Expression;
} tParsAttribute;
```

Example in Modula-2:

```
TYPE tParsAttribute = RECORD
   CASE : INTEGER OF
   | 0: Scan            : tScanAttribute;
   | 1: Statement       : tTree;
   | 2: Expression      : tValue;
   END;
END;
```

Example in Ada:

```
type tParsAttribute (Nonterm: Integer := 0) is record
   case Nonterm is
   when 0 => Scan       : Scanner.tScanAttribute;
   when 1 => Statement  : tTree;
   when 2 => Expression : tValue;
   when others => null;
   end case;
end record;
```

Example in Eiffel (usually as a separate class file):

```
class StatementAttribute
inherit Attribute
feature
   Statement : tTree
end
```

Example in Java:

```
GLOBAL {
# define yyParsAttribute ParsAttribute
}
```

and then in a separate class file or in the EXPORT section:

```
class ParsAttribute {
   public Tree tree;
}
```

The values of the attributes are computed within semantic actions. There are two possibilities to denote the access of attributes: numeric access and symbolic access.

Numeric access uses the pseudo variables $1, $2, ... to denote the attributes of the right-hand side symbols. Terminals, nonterminals as well as semantic actions have to be counted from left to right starting at the number one in order to derive the indexes. The pseudo variable $$ denotes the attribute of the left-hand side. Usually $$ is computed depending on $1, $2, ... etc. This flow of information from the right-hand side to the left-hand side of a rule is characteristic for synthesized attributes.

The above numbering scheme is valid for semantic actions placed at the end of right-hand sides as well as for semantic actions within a right-hand side. Semantic actions within a right-hand side may only access attributes of preceding symbols, or in other words, symbols to their left. For semantic actions within a right-hand side a second numbering scheme can be used as well. The

indexes start at zero for the immediately preceding symbol and decrease from right to left: $0, $-1, $-2, ...

Symbolic access uses identifiers instead of numbers. Symbol names or short names can be introduced by appending the character '$' and an identifier to the terminal or nonterminal grammar symbols. Symbolic names can also be appended to semantic actions, both, conditional or unconditional ones, if these are located within the right-hand side of rules. Symbolic access consists of the character '$' immediately followed by a symbol name. Again, semantic actions within a right-hand side may only access attributes of preceding symbols or actions.

Example:

```
X : a$a { A; }$N b$d { B; } c { C; } .
```

The transformation of this rule results in the following "pure" rules:

```
X : a 1_X_Act_2 b 1_X_Act_4 c { C; } .
1_X_Act_2 : { A; } .
1_X_Act_4 : { B; } .
```

The identifiers a, N, and d are symbolic names for a, { A; } alias 1_X_Act_2, and b. The following table lists for every symbol and action of the rule the possibilities for accessing its attributes depending on whether the access is located in semantic action A, B, or C. Within A and B, $$ does not refer to X but to the left-hand sides of the extra rules added for A and B. The attributes of the symbol c and the action B can be accessed only in the numeric style with $4 and $5 because symbolic names are not introduced for c and B.

	A	B	C
X	-	-	$$
a	$1, $0, $a	$1, $-2, $a	$1, $a
{ A; }	$$	$2, $-1, $N	$2, $N
b	-	$3, $ 0, $d	$3, $d
{ B; }	-	$$	$4
c	-	-	$5
{ C; }	-	-	-

If the type *tParsAttribute* is a union or a struct type as in the example above, the attribute access may be followed by selectors for members or fields.

Example:

```
expression: '(' expression ')'            { $$.Value = $2.Value;            } .
expression: expression   '+' expression   { $$.Value = $1.Value + $3.Value; } .
expression: expression$l '-' expression$r { $$.Value = $l.Value - $r.Value; } .
expression: integer                       { $$.Value = $1.Scan.Value;       } .
```

2.10. Syntactic and Semantic Predicates

Grammar rules can optionally be augmented with predicates. Like the specification of precedence and associativity for operator tokens, predicates are used only in case of LR conflicts as instructions for conflict resolution. There are semantic predicates and syntactic predicates. All predicates are introduced by a question mark '?' and they are written at the end of the right-hand side of a

grammar rule. A semantic predicate consists of a condition enclosed in curly brackets '{' and '}'. A syntactic predicate consists of an identifier or a string denoting a terminal or a nonterminal grammar symbol. The predicates can be negated by using the prefix operator '-':

```
X : X1 ... Xn ? { condition } .
X : X1 ... Xn ? terminal .
X : X1 ... Xn ? nonterminal .
X : X1 ... Xn ? - { condition } .
X : X1 ... Xn ? - terminal .
X : X1 ... Xn ? - nonterminal .
```

A predicate may also appear within the right-hand side of a grammar rule. Like in the case of semantic actions, the grammar is internally transformed by moving all predicates to the end of right-hand sides. Again, this is done by the introduction of new nonterminals and new rules with empty right-hand sides.

Example:

The rule	`X : X1 ... ? predicate ... Xn .`
is turned into	`X : X1 ... Y ... Xn .`
and	`Y : ? predicate .`

As already mentioned, predicates are used only in case of LR conflicts. LR conflicts are explained in detail in the chapters 4 and 5. A rule with a predicate is reduced only if the predicate yields true during run time. A semantic predicate yields true if the condition yields true. A syntactic predicate with a terminal yields true if the current lookahead token is equal to this terminal. A syntactic predicate with a nonterminal yields true if the following tokens can be correctly parsed as this nonterminal. The operator '-' negates the value of a predicate. In this case a rule would be reduced only if the predicate yields false. The last form of a predicate constitutes trial parsing or backtracking. It is described further in chapter 5. The nonterminal can be chosen freely. It can either be the nonterminal that follows anyway, a different nonterminal from the grammar, or a nonterminal that is not used in the "pure" grammar at all. In the latter case an artificial nonterminal is introduced just for the purpose of trial parsing.

As a first example we present a semantic predicate. The rule will be recognized if either there is no conflict involved or there is a conflict and the condition evaluates to true. We assume that IsTypedefName is a boolean function which looks up a certain property of an identifier in a symbol table.

```
TYPEDEFname : IDENTIFIER$i ? { IsTypedefName ($i.Scan.Ident) } .
```

At a first glance it seems that predicates can be executed only at the end of a rule or in other words in combination with a reduce action. Predicates can be connected with shift actions, too. The trick is to introduce extra nonterminals with an empty rule and to insert it into the right-hand side of a rule where the predicate should be checked. In the following example the nonterminal is_decl is introduced only to initiate trial parsing at the beginning of a rule.

```
declaration : is_decl declaration_specifier declarator ';' .
is_decl     : ? declaration .
```

A third possibility for predicates is to write them within right-hand sides of rules. The previous example can be rewritten as follows:

```
declaration : ? declaration declaration_specifier declarator ';' .
```

This rule has the same meaning as the two rules in the previous example. Although it might seem like a left-recursion which does not terminate, this is not the case. The reason is that predicates are evaluated only in case of LR conflicts. Let's assume the above rule is involved in a conflict. Then trial parsing would be started with declaration as start symbol. But the above rule would solely define the start state for this trial parse and therefore the start state would not contain a conflict. As a consequence the predicate is not used during trial parsing (at least at the start state) and there is no danger of an infinite recursion of trial parses.

A possible condition to be used in a semantic predicate could involve information about the last rule or more precisely the last nonterminal recognized by the parser. For this purpose the parser provides a variable called

```
yyNonterminal
```

and a set of named constants such as:

```
yyNTdeclaration
yyNTstatement
yyNTexpression
```

A constant is generated for every nonterminal by prefixing its name with *yyNT*. The variable yyNonterminal holds the nonterminal of the left-hand side of the rule which has been reduced last. This variable can be accessed in semantic predicates or in semantic actions and this may be of good use in determining whether a trial parse needs to be done. This feature was termed *reduction tracking* in [Mer93].

It must be noted, that semantic predicates should not and can not be used for tasks of the semantic analysis in a compiler. First, one can not rely on whether a semantic predicate is actually executed, because it is used only in case of LR conflicts. Second, the purpose of this construct is to be able to control parsing depending on information from a symbol table or some other global data structure. The emphasis is still on parsing or syntactic analysis. Or in other words, the main purpose of a parser is still to perform syntactic analysis and not to perform semantic analysis. This does not prevent semantic analysis to be performed during parsing. However, this should be done just by appropriate code in the semantic actions of the rules. This would also be the right place for the formulation of context conditions. The check of context conditions is a classical task of semantic analysis and it should not be mixed up with semantic predicates in a grammar whose sole purpose is to support syntactic analysis.

3. LALR(1) and LR(1) Grammars

The parser generator *Lark* processes LALR(1), LALR(2) as well as LR(1) grammars. The following is valid if the number of lookahead tokens is one, which is the default. With option -0 an LALR(1) parser is generated which is based on a so-called LR(0) automaton. This is the default behaviour. With option -1 an LR(1) parser is generated which is based on a so-called LR(1) automaton. With options -01 an LALR(1) parser is generated which is extended locally to LR(1) where necessary using an algorithm similar to the lane-tracing algorithm published by Pager [Pag77a, Pag77b]. A text book example of an LR(1) grammar which is not LALR(1) is the following:

```
TOKEN a b c d
RULE
   S : A a | b A c | B c | b B a .
   A : d .
   B : d .
```

The commands 'lark' or 'lark -0' yield the following messages:

```
Warning     reduce reduce conflict implicitly repaired at state: 3
            on lookahead: a c
Warning     grammar is not LALR(1)
```

The commands 'lark -1' or 'lark -01' yield:

```
Information grammar is LR(1)
```

In general, an LR(1) automaton has many more states than an LR(0) automaton and the generation effort is considerably higher. Therefore, the default settings for the options should be given preference over the explicit use of the options -1 or -01. The options -01 will solve as many LR conflicts as will option -1. In general, this is done with less consumption of time and memory during parser generation and it results in a smaller parser.

Although the class of LR(1) grammars is in theory larger than the class of LALR(1) grammars, it should not be expected that an LR(1) parser generator solves many more conflicts than an LALR(1) parser generator. First, all shift-reduce conflicts found during LALR(1) parser generation are also detected during LR(1) parser generation. There is only a chance that the number of reduce-reduce conflicts is decreased. Second, as the number of states may increase, either because of the lane-tracing algorithm or because of the request for an LR(1) automaton with option -1, it is possible that the number of states with conflicts may increase, too. This does not mean that any conflicts caused by new reasons are added, but the existing conflicts may be spread over to states created additionally. For large grammars with many reduce-reduce conflicts it can be advantageous to use option -0 because the lane-tracing algorithm can tend to the generation of a full LR(1) parser without solving more conflicts than in the LALR(1) case. According to the current practical experience, grammars that are not LALR(1) but which are LR(1) are rarely found.

4. Ambiguous Grammars

In some cases language definitions are ambiguous or it may be more convenient to describe a language feature by ambiguous rules than by unambiguous ones. In general, the structure of input according to an ambiguous grammar can not be recognized unmistakable, because there are several solutions. Ambiguous grammars do not fall into the classes of LALR(1) or LR(1) grammars. Without additional information, ambiguous grammars can not be processed sensibly by *Lark*. This section describes how to solve many ambiguity problems.

The classical example which leads to an ambiguous grammar is the *dangling else* problem which is present for example in Pascal and in C. Suppose a grammar contains two rules for IF statements such as the following:

```
statement : 'IF' expression 'THEN' statement .
statement : 'IF' expression 'THEN' statement 'ELSE' statement .
```

Analyzing the input

```
IF b THEN IF c THEN d ELSE e
```

it is not clear whether the ELSE belongs to the first or to the second IF THEN. Another typical example is the definition of expressions by rules like the following:

```
expression : expression '*' expression .
expression : expression '+' expression .
expression : '(' expression ')' .
expression : identifier .
```

Given a grammar containing the above rules *Lark* would produce a message saying the grammar contains a conflict and that it is not LR(1).

```
Warning     shift reduce conflict implicitly repaired at state: 264
            on lookahead: ELSE
Warning     grammar is not LR(1)
```

Before we describe the meaning of this message and what to do in such a case we have to say briefly how the generated parser works.

4.1. LR Conflicts

The generated parser is a stack automaton controlled by a parse table. The automaton is characterized by the contents and the administration of a stack and a set of states. A state describes a part of the input already analyzed. The operation of the automaton consists of the repeated execution of steps. A step is the execution of an action and the transition from the actual state to another one. The steps are controlled by the parse table which basically implements a transition function, mapping a state and the next input token to an action:

$$\text{Table} : \text{State} \times \text{Token} \rightarrow \text{Action}$$

There are primarily two actions: The action *shift* means to read or consume an input token. The action *reduce* is used when a rule has been recognized and it means to imaginarily replace in the input the right-hand side of the recognized rule by its left-hand side.

Given an ambiguous grammar, the above transition function can not be computed, because the function would be ambiguous, too. For some table entries characterized by a pair (state, token) there would be several different actions. Two cases can arise: If a table entry could contain a shift action as well as a reduce action we have a *shift-reduce conflict*. If a table entry could contain two reduce actions concerning different rules we have a *reduce-reduce conflict*. In general, not only two actions are involved in a conflict but an arbitrary number. If a conflict is detected, its kind, the state number, and the token involved are reported. In the above message, the parser generator reports a shift-reduce conflict at a state with the internal number 264. This conflict arises if the next input token is ELSE. The next input token is also called lookahead token or just lookahead.

4.2. Conflict Resolution

In case of a conflict, *Lark* applies the following steps in order to construct an unambiguous transition function. For all rules involved in a conflict a precedence and associativity is determined, if possible. The rules indicating a shift action receive the precedence and associativity of the token to be shifted. The rules indicating a reduce action retain their own precedence and associativity values. These are either determined by the right-most operator in the rule or by an explicitly given PREC clause. The latter dominates any existing operators (see also section 2.5). Now, several cases can be distinguished depending on whether all rules involved in the conflict have precedence and

associativity values or whether there are rules with (syntactic or semantic) predicates.

If the conflict can be resolved without using predicates then it is resolved statically during generation time of the parser, otherwise it is resolved dynamically during the run time of the parser.

4.2.1. Explicit Repair

If all involved rules have precedence and associativity values, the conflict resolution proceeds as follows:

- In case of a shift-reduce conflict and rules with different precedences, the action of the rule with highest precedence is preferred. If all rules have the same precedence then the associativity (which must be the same for all rules) is considered: Left associativity selects the reduce action and right associativity selects the shift action. Otherwise no associativity is specified. Now, if there are rules with predicates then "dynamic repair" is performed as described below, else an error message is reported.
- In case of a reduce-reduce conflict, the rule with the highest precedence is reduced. If there are several rules with the same highest precedence and there are rules with predicates then "dynamic repair" is performed as described below, else an error message is issued.

These conflict solutions are classified as *explicit repair* as long as "dynamic repair" is not performed. They are reported as informations.

4.2.2. Implicit Repair

If there is at least one rule without precedence and associativity and none of the rules has a (syntactic or semantic) predicate, the conflict is resolved as follows:

- In case of a shift-reduce conflict, the shift action is preferred.
- In case of a reduce-reduce conflict, the rule given first is reduced.

These conflict solutions are classified as *implicit repair*. They are reported as warnings.

4.2.3. Dynamic Repair

If there are rules with a (syntactic or semantic) predicate and "explicit repair" either is not applicable or it failed then the conflict is resolved dynamically during the run time of the parser in the following way:

- The predicates are checked in a certain order until one yields true. Then the associated rule is recognized (reduced). If all predicates yield false then the conflict is resolved similar to the "implicit repair" strategy described above: In case of a shift-reduce conflict, the shift action is preferred. In case of a reduce-reduce conflict, the first rule without a predicate is reduced. If all rules have a predicate then the first rule with a predicate is reduced.
- A semantic predicate of the form "? { condition }" yields true if the evaluation of the condition yields true.
- A syntactic predicate of the form "? terminal" yields true if the lookahead token is equal to this terminal.
- A syntactic predicate of the form "? nonterminal" yields true if the tokens not processed so far can be derived from this nonterminal. This initiates a so-called *trial parse* which is explained further below.
- A predicate with a negation operator ’-’ (form "? - predicate") reverses the result of the previous definitions.

The conflict solutions performed during run time are classified as *dynamic repair*. They are reported as warnings by the parser generator.

4.2.4. Trial Parsing

The condition in a semantic predicate may access attributes in the same way as semantic actions. It is recommended to use solely read-only access of attributes and of global variables in a condition. In general, side-effects are possible in a condition - however, this is problematic and deserves special consideration: First, one can not rely on the execution of those side-effects, because the condition is evaluated only in case of LR conflicts. Second, the rule could be reduced at several states of the parser. Some of these states could have a conflict and others don't. Then the condition would be evaluated sometimes in order to try to recognize the rule and sometimes not. Third, no mechanism is provided to automatically undo any side-effects - this would fall into the responsibility of the user.

A *trial parse* checks if the tokens not processed so far can be derived from a nonterminal. This is accomplished by recursively starting a parse which uses this nonterminal as start symbol. If the trial parse succeeds the predicate yields true. If the trial parse detects syntax errors then these errors are not reported and the predicate yields false. In both cases the stream of tokens is backed-up to the position before starting the trial parse. This method can also be called backtracking parsing. It is user controlled backtracking because the user can explicitly specify when to apply a trial parse by adding a predicate and what to check for by the selection of an appropriate nonterminal. Trial parsing or backtracking may be nested. This is the case if during trial parsing an other conflict is resolved using trial parsing.

The semantic actions deserve special consideration with respect to trial parsing. In general, semantic actions executed during trial parsing should be undone because only the effect of the semantic actions executed during the final parse is desired. However, to undo arbitrary semantic actions is expensive and more or less impossible. Therefore, two kinds of semantic actions are distinguished: conditional semantic actions and unconditional semantic actions. Conditional semantic actions are enclosed in curly brackets '{' and '}', unconditional semantic actions are enclosed in edged brackets '[' and ']'. Unconditional semantic actions are executed by default during all parsing modes: trial, standard, buffer, and reparse. Conditional semantic actions are executed by default in the parsing mode standard, they are not executed in mode trial, and during the modes buffer and reparse their execution is under user control. In general, the execution of semantic actions can be additionally controlled by an action flag and so-called selection masks. The section about "Reparsing" describes the details.

The "usual" semantic actions representing the result or output of parsing are conditional semantic actions and should therefore be enclosed in '{' and '}'. The unconditional semantic actions enclosed in '[' and ']' are primarily provided for attribute computations during trial parsing and for the explicit execution of YYACCEPT or YYABORT statements. These latter statements are currently available for the target languages C, C++, Java, and Modula-2. They can be used for the premature termination of standard or trial parsing with or without errors. Attribute computations might be necessary during trial parsing if attribute values are used in semantic predicates. In general arbitrary actions including those causing side-effects are possible. However, no mechanism is provided to automatically undo those side-effects in case a trial parse fails. It is in the users responsibility to either undo side-effects or to cope with them otherwise. It is recommended not to use side-effects in unconditional semantic actions.

4.3. Examples

The dangling else ambiguity already mentioned in chapter 4, is usually solved correctly by the implicit repair strategy. Therefore, additions to the grammar rules are not necessary:

```
statement : 'IF' expression 'THEN' statement .
statement : 'IF' expression 'THEN' statement 'ELSE' statement .
```

However, an implicitly repaired conflict leads to a Warning message:

```
Warning     shift reduce conflict implicitly repaired at state: 264
            on lookahead: ELSE
Warning     grammar is not LR(1)
```

The dangling else problem can also be repaired explicitly. We regard THEN and ELSE as operators and define precedence levels for them:

```
PREC
   NONE 'THEN'
   NONE 'ELSE'
```

Now the conflict is repaired explicitly and this leads to an Information message:

```
Information shift reduce conflict explicitly repaired at state: 264
            on lookahead: ELSE
Information grammar is LALR(1) after explicit repair
```

Grammar rules for expressions can be written mainly in two styles: First, separate nonterminals are used for different levels of precedence such as for example expression, term, and factor (see example in section 2.7). Usually this kind of rules do not lead to an ambiguity. However, a parser for these rules is relatively inefficient, because in order to recognize a constant as a possible expression as many reduce actions are necessary as there are levels of precedence (e. g. more than 10 in C).

Second, only one nonterminal is used for the description of expressions (see example in chapter 4). A disadvantage of this style is that the rules are ambiguous and cause several LR conflicts. However, the conflicts can easily be resolved by the explicit specification of precedence and associativity for every operator as shown below. The advantage of this style is its efficiency because every construct in an expression can be recognized with a single reduce action.

Example: Precedence and Associativity for Pascal Operators

```
PREC
   LEFT '=' '<>' '<' '<=' '>' '>=' 'IN'
   LEFT UNARY_OPERATOR
   LEFT '+' '-' 'OR'
   LEFT '*' '/' 'DIV' 'MOD' 'AND'
   LEFT 'NOT'
   NONE 'THEN'
   NONE 'ELSE'
```

The operator UNARY_OPERATOR is not a Pascal operator. It is added in order to define a separate precedence level for the unary operators '+' and '-'. This separate precedence level is associated with the grammar rules for unary operators with a PREC clause (see section 2.5).

Example: Description of Expressions for Pascal (Excerpt)

```
expression : '+' expression PREC UNARY_OPERATOR
           | '-' expression PREC UNARY_OPERATOR
           | expression '+' expression
           | expression '-' expression
           | variable
           | constant
           .
```

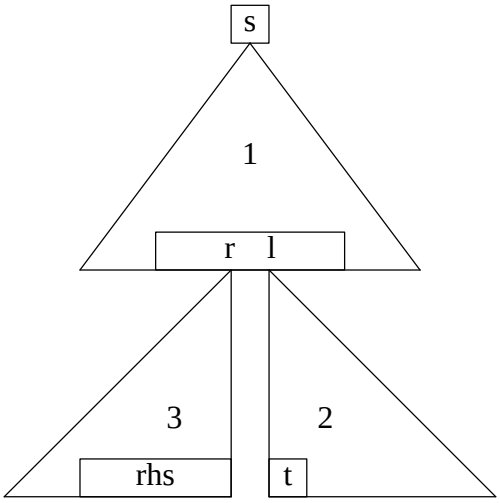
More severe LR conflicts can be solved with predicates. Examples for the syntax of predicates can be found in section 2.10. A larger example for dynamic conflict repair is described in section 5.3.

5. Explanation of LR Conflicts

If there are conflicts in the grammar and the options -v (verbose) or -w are set, then the reasons for the conflicts are explained. While option -v selects the explanation of all conflicts, option -w selects the explanation of implicitly and dynamically repaired conflicts and it suppresses the explanation of explicitly repaired conflicts. The explanation is written to a file whose name consists of the name of the input file with the suffix *.dbg*. The name is *Parser.dbg* when the grammar is read from standard input. The technique for the explanation of conflicts follows the method published in [DeP82].

5.1. Derivation Trees

For every state with conflicts and for every so-called situation involved in a conflict, a fragment of a derivation tree is printed. Situations are also called items in the parsing literature. We will use both terms as synonyms. A situation consists of a grammar rule, a lookahead token, and a position. A position describes how far a rule has been recognized in this state. It is indicated by a dot character in the right-hand side of the rule. The derivation tree explains how a lookahead token and a rule can interfere. The derivation tree has three parts as shown in Figure 1:



s: root of part 1 (start symbol)
l: root of part 2
r: root of part 3
t: lookahead token
rhs: right-hand side of rule

Fig. 1: Structure of the Derivation Tree used to explain an LR Conflict

- The first part describes the derivation from the start symbol s of the grammar to an intermediate rule. Two neighbouring symbols r and l in this intermediate rule are the roots of the other two parts (subtrees).
- The second part uses the right one of those two symbols as root (l). It describes the derivation of the lookahead token. The lookahead token is the left-most token in the last rule of this part (subtree).
- The third part uses the left one of those symbols as root (r). It describes the derivation of the rule.

The three parts of a derivation tree are printed in an ASCII representation one after the other.

5.2. Explicit and Implicit Repair

In this section we describe the explanation for conflicts that can be repaired either explicitly or implicitly. The next section deals with dynamically repaired conflicts. For the description of the structure of the conflict explanation we will use the example of the *dangling else* conflict mentioned in section 4. We have added line numbers to the left as points of reference. The explanation of a conflict consists of two parts: The first part contains the derivation trees presented above and the second part contains a summary that describes the resolution of the conflict. In our example, the first part starts at line 1 and the second parts start at line 22.

Line 1 gives two numbers of a state with a conflict: An external number followed by an internal number enclosed in parentheses. The internal number of a state is used within the parser generator. It is in general different from the external number of the state used in the generated parser. The

Example: Explanation produced for the "dangling else" conflict

```
 1  State  192 ( 264): derivation trees
 2  ----------------------------------
 3
 4          shift reduce conflict implicitly repaired
 5
 6  .program _EOF_
 7   'PROGRAM' Identifier prog_params ';'.block '.'
 8   label const type var proc 'BEGIN'.statement_seq 'END'
 9  .statement
10   'IF' expr 'THEN'.statement 'ELSE' statement
11  |  'IF' expr 'THEN' statement.'ELSE' statement
12   'IF' expr 'THEN' statement.
13
14  reduce  228  statement:  'IF' expr 'THEN' statement. {'ELSE'} ?
15
16   ...
17  .statement
18   'IF' expr 'THEN' statement.'ELSE' statement
19
20  shift   229  statement:  'IF' expr 'THEN' statement.'ELSE' statement ?
21
22  State  192 ( 264): summary
23  --------------------------
24
25  ignore reduce  228  statement:  'IF' expr 'THEN' statement. {'ELSE'}
26  retain shift   229  statement:  'IF' expr 'THEN' statement.'ELSE' statement
```

difference between these numbers arises from the fact that some states become useless after optimizations and a consecutive sequence of external numbers is used for the states in the generated parser. Note, that even states with a conflict can be removed during optimization. Those states are reported with an external number of zero.

Line 4 describes the type of the conflict (shift reduce conflict in our example) and which strategy the generator used to resolve it (implicit repair).

The lines 14 and 20 contain the two situations or items of this state which give rise to the conflict. In both cases the lookahead token is 'ELSE'. Upon seeing the lookahead token 'ELSE' the first situation leads to the action "reduce the rule `statement: 'IF' expr 'THEN' statement`" whereas the second situation leads to the action "shift the token `'ELSE'`". A shift action would mean to continue the analysis of the rule "`statement: 'IF' expr 'THEN' statement.'ELSE' statement`" by moving the dot '.' behind the token 'ELSE'. The numbers 228 and 229 refer to the line numbers of the rules in the grammar. The dots '.' within the right-hand sides of the rules describe how far the analysis of the rules has proceeded. The token to the right of the dot '.' is the lookahead token. In case of a reduce action, the dot is at the end of the right-hand side and the lookahead token is not part of the right-hand side. The lookahead token is determined by the possible context of the rule. It is enclosed in curly brackets '{' and '}' in order to express its origin from the context.

The lines above the situations explain why these situations are in the same state and how the lookahead token for reduce situations can follow a rule. The explanation for reduce situations consists of a derivation tree with three parts as mentioned in section 5.1. The first part of the derivation starts at the start symbol of the grammar (line 6: program) and leads to a rule where the derivation for the lookahead starts (line 10). Every line contains the right-hand side of a grammar rule. The dot in a line describes how far the analysis has proceeded. The symbol after a dot is the left-hand side for the rule in the next line. In the last line of the first part of the derivation (line 10) two symbols are important. The symbol after the dot (statement) is the left-hand side or root for the third part of the derivation and the next symbol ('ELSE') is the left-hand side or root for the second part of the derivation which explains the lookahead token.

In the above example, the second part of the derivation consists just of one line (line 11). It is marked with the character '|' at the left margin. It starts with the same right-hand side as the previous line, only the dot has moved one symbol to the right. In our example we have reached a terminal symbol which constitutes the lookahead token ('ELSE'). If this symbol is a nonterminal then a derivation consisting of several lines would be printed. In the last line of the second part of the derivation the dot is always in front of a terminal symbol and thus explains the origin of the lookahead token. In general there can be several contexts that contribute the lookahead token. In this case several different derivations will be produced, one for every context.

The third part of the derivation is contained in line 12. The nonterminal after the dot of the first part of the derivation (line 10: statement) is the left-hand side or root for this part of the derivation. In general the third part of the derivation can extend over several lines. The last line (line 12) represents the conflicting situation or item.

For shift situations only the first part of a derivation is printed because there is no lookahead token from a context involved in this case. This derivation starts from the start symbol and leads to a situation where the dot describes the current state of the analysis (line: 18). It can be observed that many derivations share a common beginning. In order to present only information of relevance, common beginnings of derivations are not repeated. Instead the characters '...' are printed. Only the last line of common beginnings is displayed (line 17 is equal to line 9). Therefore the characters '...' in line 16 stand for the same information as is contained in lines 6 to 8.

Example: Grammar with dynamic resolution of LR conflicts

```
RULE
compound_statement      : '{' declaration_list statement_list '}'
                        .
declaration_list        :
                        | declaration_list declaration
                        .
statement_list          :
                        | statement_list statement
                        .
declaration             : ? declaration declaration_specifier declarator ';'
                        .
declaration_specifier   : TYPEDEFname [ printf ("trial action\n"); ]
                                      { printf ("final action\n"); }
                        .
declarator              : IDENTIFIER  [ printf ("trial action\n"); ]
                                      { printf ("final action\n"); }
                        | '(' declarator ')'
                        | declarator '[' ']'
                        | compound_statement
                        .
statement               : expression ';'
                        .
expression              : expression '++'
                        | TYPEDEFname '(' expression ')'        /* cast */
                        | IDENTIFIER  '(' expression ')'        /* call */
                        | IDENTIFIER
                        .
TYPEDEFname             : IDENTIFIER$i ?
                          { GetString ($i.Scan.Ident, name), isupper (name [0]) }
                        | IDENTIFIER ? IDENTIFIER
                        .
```

The information about an LR conflict is terminated by a summary (lines 22 to 26). The summary explains how the conflict was resolved. Conflicts are solved by splitting the situations into two sets of so-called ignored situations and retained situations. The actions induced by the set of ignored situations are ignored. The retained situations are consistent because they induce exactly one action. This action will be taken by the generated parser. In the example the parser will shift the lookahead token 'ELSE' in state 264. Again, the numbers 228 and 229 refer to the line numbers of the rules in the grammar.

5.3. Dynamic Repair

In this section we describe the explanation for conflicts that are repaired dynamically during the run time of the generated parser. We will use a tiny excerpt from a C++ grammar adapted from [Mer93] that solves parsing problems of C++ using semantic predicates and trial parsing (backtracking).

First, there is the typedef problem (also present in C) where it is to decide whether an IDENTIFIER token denotes a typedef name or an identifier. Unlike other solutions, we do not make this decision in the scanner but in the parser. For this reason the first right-hand side of the grammar rule for the nonterminal TYPEDEFname has been augmented with a semantic predicate. Its condition simulates a symbol table lookup that checks whether an IDENTIFIER should be treated as typedef name. Our simulation treats IDENTIFIERs as typedef names when the first letter is a capital letter.

Second, there is another ambiguity in the grammar rising from the new style of casts for expressions. The token sequence

```
TYPEDEFname ( IDENTIFIER ) ;
```

can be recognized as a declaration or as a statement. C++ solves this problem with the requirement that in case of such an ambiguity the construct shall be treated as a declaration. One solution for this requirement is to use trial parsing before every declaration. If trial parsing using the nonterminal declaration succeeds then we are sure to have a declaration and can perform final or standard parsing for a declaration. Otherwise we do not parse it as a declaration in which case the parser would continue to analyze a statement_list. Trial parsing is triggered by the construct "? declaration" on the right-hand side of the rule for the nonterminal declaration. The example grammar contains four LR conflicts which are reported as follows:

```
Information reduce reduce conflict dynamically repaired at state: 7
            on lookahead: IDENTIFIER
Information reduce reduce conflict dynamically repaired at state: 8
            on lookahead: IDENTIFIER ( {
Information shift reduce reduce conflict dynamically repaired at state: 35
            on lookahead: (
Information shift reduce reduce conflict dynamically repaired at state: 36
            on lookahead: (
Information reduce reduce conflicts: 2
Information shift reduce reduce conflicts: 2
```

The last two states with conflicts contain a shift reduce conflict as well as a reduce reduce conflict for one lookahead token. These kinds of conflicts are classified as *shift reduce reduce conflicts*. We will discuss the explanation for state 36 in this chapter. Again, we have added line numbers to the left as points of reference.

The lines 1 to 27 contain the derivation trees for the rules involved in the conflict. As three rules are involved, three derivation trees are produced. The interesting part is found in the summary starting at line 32 where the dynamic conflict resolution is described. Line 32 states the fact that a dynamic decision is used if the parser is in state 36 and the lookahead token is ’(’. The different solutions of conflicts performed dynamically during run time are numbered. In this case the dynamic decision scheme has number 4. First, the predicate of the grammar rule in line 50 will be checked, which is a semantic condition (line 33). Then, the predicate of the grammar rule in line 51 will be checked. The latter is a syntactic terminal predicate that tests whether the lookahead token is ’(’ (line 35). This syntactic predicate makes no sense in C++ - it has been added to construct a state that leads to a dynamic decision where more than one predicate is checked. The predicates are checked in the given order until one yields true. Then the associated rule is reduced. If none of the predicates yields true, then a default action is performed. The possible default actions are listed after line 36. They have to be interpreted as in the case of explicit or implicit repair. The retained action is taken. In the example the default action is to shift the token ’(’.

5.4. Explanation of Differences

The options -v or -w produce the explanation of all applicable conflicts, by default. In extreme cases when there are many conflicts these explanations can be rather bulky and therefore tedious to inspect. The additional option -D restricts the set of explained conflicts to those which are new with respect to the previous run of the parser generator. This option works as follows:

Example: Explanation produced for a dynamically repaired LR conflict

```
 1  State   19 (  36): derivation trees
 2  ----------------------------------

 4          shift reduce reduce conflict dynamically repaired

 6  .compound_statement _EOF_
 7   '{' declaration_list.statement_list '}'
 8   statement_list.statement
 9  .expression ';'
10   TYPEDEFname '('.expression ')'
11  .TYPEDEFname '(' expression ')'
12  |  TYPEDEFname.'(' expression ')'
13   IDENTIFIER.

15  reduce   49  TYPEDEFname:  IDENTIFIER. {'('} ?

17   ...
18  |  TYPEDEFname.'(' expression ')'
19   IDENTIFIER.

21  reduce   51  TYPEDEFname:  IDENTIFIER. {'('} ?

23   ...
24   TYPEDEFname '('.expression ')'
25   IDENTIFIER.'(' expression ')'

27  shift    46  expression:  IDENTIFIER.'(' expression ')' ?

29  State   19 (  36): summary
30  --------------------------

32  dynamic decision 4 on lookahead '(':
33  check predicate in line 50: ? { GetString ($i.Scan.Ident, name),
34                                  isupper (name [0]) }
35  check predicate in line 51: ? '('
36  default:
37  ignore reduce   49  TYPEDEFname:  IDENTIFIER. {'('}
38  ignore reduce   51  TYPEDEFname:  IDENTIFIER. {'('}
39  retain shift    46  expression:  IDENTIFIER.'(' expression ')'
```

When *lark* is invoked the first time with the option -D it produces an additional file with the suffix *.cft* (for conflict) which contains an internal representation of all conflicts. During subsequent runs with the option -D this file is used to determine the set of new conflicts. Explanations in the file with the suffix *.dbg* are produced for new conflicts, only. The file with the suffix *.cft* is updated to reflect the current set of conflicts. Additionally, an output file with the suffix *.dlt* (for delta) is produced which summarizes the differences of the grammar and the set of conflicts with respect to the previous run.

The lines in a delta file (see example) have the following meaning: The characters '-' and '+' in column one indicate omissions and additions of items. The items under consideration are given by the next word in every line which can be one of: Terminal, Nonterm, Rule, or Conflict. The following information in the line describes the entry. At the end of the file a summary lists the numbers of the omitted and added items.

This feature can be used for several purposes. First, it reduces the set of explained conflicts and thus easies the effort involved in conflict inspection. Second, the summary of differences in the grammar allows for the validation of the last changes. Third, this feature can be used to compare

Example: Contents of the delta file Parser.dlt (excerpt)

```
+ Terminal STORE
+ Terminal ANSI

- Nonterm  5_accept_i_Trial_2
- Nonterm  5_accept_Trial_2

+ Nonterm  special_names_l
+ Nonterm  page_o

- Rule     set: SET set_l TO ON on_off_l
- Rule     set: SET set_l TO OFF on_off_l
- Rule     on_off_e: name_l TO ON
- Rule     on_off_e: name_l TO OFF

+ Rule     set: SET set_l To ON on_off_l
+ Rule     set: SET set_l To OFF on_off_l
+ Rule     on_off_e: name_l To ON
+ Rule     on_off_e: name_l To OFF
+ Rule     use: USE For exception_or_error
+ Rule     use: USE For exception_or_error On use_l

+ Conflict State 320 (555) on lookahead {'SPECIAL-NAMES' }
+ Conflict State 594 (1029) on lookahead {name }
+ Conflict State 654 (1092) on lookahead {NEXT }

- Conflict State 986 (1689) on lookahead {name }

Summary
-------

+ Terminals       2
- Nonterminals    2
+ Nonterminals    2
- Rules           4
+ Rules           6
- Conflicts       1
+ Conflicts       3
```

arbitrary grammars. The file with the suffix *.cft* supplied for a run of *lark* with option -D has not to be necessarily the output of the previous run. The user can copy a conflict file from any run in the past into the current directory. This file can be from the current project or even a different one. It is just necessary that this file has been saved somewhere so that *lark* does not overwrite it. This way it is possible to get a summary of changes that have been carried out in several small steps or to compare grammar describing different dialects of a language. The comparison for grammars is done on the base of grammar rules only - semantic actions are not taken into account.

6. Reparsing

In some cases it might be necessary to parse a segment of the input twice or even several times. *Lark* supports this with the so-called *reparsing* feature. Mainly two aspects have to be regarded: The tokens have to be buffered and the execution of semantic actions has to be controllable. Buffering of the tokens avoids rescanning of the source and supplies the tokens for the second or later passes over the input. Usually, just the last pass over a segment of the input should execute the so-called final semantic actions which generate an internal representation of the input such as a syntax tree. However, previous passes should be able to execute non-final semantic actions allowing for example to compute symbol-table information that can be used in semantic predicates in order to guide

parsing. Only the influence of semantic predicates on parsing motivates reparsing because this way a different parsing result can be achieved from additional passes.

6.1. Parsing Modes

In combination with the existing trial parsing feature several modes of the parser can be distinguished. The following table lists the modes and characterizes every mode by the associated behaviour of the parser:

Mode	Abbreviation	Buffer	GetToken	Semantic Actions	Uncond. Actions	Error	Abort	Start	Stop
standard	S	no	scanner	yes/no	yes/no	yes/no	no	call	EOF
trial	T	yes	scanner	no/yes	yes/no	no	yes	conflict	error or accept
buffer	B	yes	scanner	yes/no	yes/no	no/yes	no	call	call or EOF
reparse	R	no	buffer	yes/no	yes/no	no/yes	no	call	EOB

The columns of the above table have the following meaning:

Column	Meaning
Mode	name of the parsing mode
Abbreviation	letter abbreviating the parsing mode
Buffer	whether the tokens are stored in a buffer
GetToken	where the tokens come from; the entry scanner means that tokens might as well come from the buffer if tokens are left in the buffer from previous parsing phases in buffer or trial mode.
Semantic Actions	whether conditional semantic actions enclosed in { } are executed
yes/no	execution depends on selection mask of the statements (see below)
Uncond. Actions	whether unconditional semantic actions enclosed in [ ] are executed
yes/no	execution depends on selection mask of the statements (see below)
Error	whether error recovery is executed upon syntax errors; this includes error messages, error repair, and error recovery.
no/yes	execution of error recovery can be requested by the user
Abort	whether parsing is aborted upon syntax errors
Start	how the mode is initiated:
call	by an explicit procedure call
conflict	implicitly in order to resolve LR conflicts during run time
Stop	when the mode terminates:
call	by an explicit procedure call
EOF	upon reaching end of file
EOB	upon reaching end of buffer
error	upon detection of a syntax error
accept	upon successful completion of parsing

6.2. Control of Reparsing

The following procedures and variables control parsing an switching between parsing modes. The details of these objects are language dependent and explained more precisely in the section about

"Interfaces". Here we give an overview using the syntax of the language C:

Procedure/Variable	Meaning
int <Parser> (void);	initiate parsing in mode standard, use the start symbol defined first in the grammar or the first nonterminal if no start symbols have been defined, return the number of syntax errors
int <Parser>2 (int StartSymbol);	initiate parsing in mode standard, use the start symbol StartSymbol, return the number of syntax errors
long BufferOn (rbool Actions, rbool Messages);	enable buffering of tokens enable/disable execution of semantic actions (see below) enable/disable reporting of error messages (see below) if mode is standard: switch to mode buffer, else ignore; return the position of the current lookahead token in the token buffer
long BufferOff (void);	disable buffering of tokens if mode is buffer: switch to mode standard, else ignore; return the position of the current lookahead token in the token buffer
long BufferPosition;	variable holding the position of the current lookahead token in the buffer
int ReParse (int StartSymbol, int From, int To, rbool Actions, rbool Messages);	initiate parsing in mode reparse, use the start symbol StartSymbol, parse from (including) buffer position From, parse up to (excluding) buffer position To, enable/disable execution of semantic actions (see below) enable/disable reporting of error messages (see below) return the number of syntax errors found during its invocation
void BufferClear (void);	signal that the contents of the token buffer can be deleted
void SemActions (rbool Actions);	enable/disable execution of semantic actions true = enable, false = disable
void ErrorMessages (rbool Messages);	enable/disable reporting of error messages true = enable, false = disable

Trial parsing as well as reparsing are implemented by recursive calls of the internal parsing procedure. All of these calls start parsing with a certain start symbol. In case of reparsing the nonterminals used as start symbols have to be declared as start symbols after the keyword START. The right context for trial parsing or the segment to be reparsed have to be derivable from the nonterminal used as start symbol. This nonterminal does not have to be one of the nonterminals already contained in the grammar. It can be a nonterminal that has been added together with grammar rules for the single purpose of trial parsing or reparsing. This allows the processing of arbitrary segments as right context or for reparsing. It should be mentioned that in both cases parsing is independent of the left context or the parsing history. This comes from the recursive activation of the parser which starts with an empty stack and a new start symbol and therefore there is no influence of the state of

the previously active parser.

The four parsing modes can be activated from other parsing modes in many combinations:

mode	can be activated from
standard	outside
trial	standard, trial, buffer, reparse
buffer	standard
reparse	standard, trial, buffer, reparse

If reparsing occurs in a nested fashion and the buffer segments of two activations overlap then one condition has to be fulfilled: The end position of the segment of the nested (younger) activation may not lie after the end position of the outer (older) activation.

The relationship between the parsing modes and the recursive activations of the internal parsing procedure is as follows: The parsing modes standard and buffer are associated with the initial activation, only. Every need for trial parsing leads to a recursive activation in parsing mode trial and every call of ReParse leads to a recursive activation in parsing mode reparse. The procedures BufferOn and BufferOff switch the mode of the initial activation between standard and buffer.

Besides by a mode, every activation is characterized by two flags that control the execution of conditional semantic actions and the reporting of error messages and by its own copy of the LOCAL section. The values of the flags are initialized according to the following table.

flag	standard	trial	buffer	reparse
Actions	true	false	argument	argument
Messages	true	false	argument	argument

The values for the parsing modes buffer and reparse are given by the arguments of the procedures BufferOn and ReParse. The values of the current invocation can be changed at any time using the procedures SemActions and ErrorMessages.

The LOCAL section is copied into the internal parsing procedure. Therefore, the variables declared in this section are local to the parsing procedure and every recursive activation leads to new copies of these local variables.

Upon return from an activation in the mode trial or reparse the state of the previous activation is reestablished. This concerns the mode, the two flags, and the LOCAL section. Additionally, a successful call of BufferOff will reset the values of the flags to the state before the last successful call of BufferOn.

The main result of reparsing will usually be some kind of internal representation such as a syntax tree. This tree or any other result is conveniently computed using the S-attribution mechanism of *Lark*. This result can be transferred from the reparsing activation of the parser to its caller by a global variable.

6.3. Semantic Actions

Unconditional (as well as conditional) semantic actions are extended by so-called *selection masks*. Groups of statements can be prefixed with a selection mask. This mask starts with a hash mark '#' and it may be followed by up to four letters out of the set S, T, B, and R. The letters specify the

parsing modes in which the following group of statements is to be executed. Example:

```
[ i = 0; #TR i = 1; #TBS i = 2; j = 0; ] { m = 0; #RS n = 0; }
```

With the variables and values defined in the generated parser this is equivalent to:

```
[
   { i = 0; }
   if ((yytrial | yyreparse) & yyControl.yyMode) { i = 1; }
   if ((yytrial | yybuffer | yystandard) & yyControl.yyMode) { i = 2; j = 0; }
]
{
   if (yyControl.yyActions) { m = 0; }
   if ((yyreparse | yystandard) & yyControl.yyMode) { n = 0; }
}
```

Only the first group of statements can be written without a selection mask. A missing selection mask in an unconditional semantic action defaults to #STBR which indicates that the statements of the group are executed in every parsing mode. A missing selection mask in an conditional semantic action indicates that the statements of the group are executed if the current action flag has the value true. Groups of statements with a selection mask behave similarly regardless whether contained in an unconditional or in a conditional semantic action. Their execution depends on the current parsing mode. The notions of unconditional and conditional semantic actions are not very precise - they are a bit historic. With the introduction of selection masks and the current action flag only the first group of statements without a selection mask in edged brackets is a real unconditional semantic action, all other groups of statements are in some way conditional ones.

```
GLOBAL {

typedef union { tScanAttribute Scan; } tParsAttribute;

int stack [10], * stack_ptr = stack;

# define push(x)        * stack_ptr ++ = x
# define pop            * -- stack_ptr
# define stack_is_empty stack_ptr == stack
}

LOCAL  { char name [256]; }

TOKEN ...

START compound_statement declaration_list

RULE

compound_statement : '{'              [ #S    BufferOn (rfalse, rfalse);
                                        #SBTR push (BufferPosition);       ]
                     declaration_list [ ReParse (yydeclaration_list, pop,
                                           BufferPosition, rtrue, rtrue);
                                        if (stack_is_empty) {
                                           BufferOff (); BufferClear (); } ]
                     statement_list
                     '}'
                   .
/* further rules as before */
```

6.4. Example

The following example is an extension of the grammar used in the chapter about "Dynamic Repair". The declaration list in a compound statement is parsed twice. An action is inserted right before the nonterminal declaration_list which enables buffering of tokens using a call of BufferOn and which records the beginning of a declaration list by inspecting the variable BufferPosition. During the first pass over a declaration list semantic actions as well as error messages are disabled. A second action is inserted immediately after the nonterminal declaration_list where the second pass is activated using a call of ReParse. Appropriate values are supplied as arguments in order to specify which segment of the buffer shall be parsed again and according to which start symbol. This time semantic actions and error messages are enabled.

This example is extremely complicated because declaration lists can be nested and because of the presence of trial parsing. The example specifies that reparsing is performed every time a declaration list is recognized regardless of the parsing mode. Therefore declaration lists are parsed again in the modes standard, trial and reparse as well as at every nesting level. Not only trial parsing and reparsing are nested but they may be triggered from each other. This may lead to numerous parsing of the same segment depending on the nesting depth and the occurrence of trial parsing. Also, a stack is necessary to store the starting positions of declaration lists at every nesting level. Only when the stack is empty it is correct to disable the token buffer and to clear it using calls of BufferOff and BufferClear.

7. Listings

Several listings with information about the grammar such as the sets of terminals and nonterminals or the generated automaton can be selected with the options -x, -y, and -z. All listings are printed on standard output. The grammar from the previous chapter is used in the examples of this chapter.

7.1. Terminals

The option -x selects a listing of all terminal symbols of the grammar. This list contains all terminals, either declared explicitly or implicitly, and shows the integer representation used in the generated parser. The scanner procedure GetToken has to return these values. The list starts always with the special token _EOF_ encoded by 0 which signals end of file.

Example:

```
Terminals

_EOF_     0
IDENTIFIER        1
'('       2
')'       3
'++'      4
';'       5
'['       6
']'       7
'{'       8
'}'       9
```

7.2. Nonterminals and Rules

The option -z selects a listing of all nonterminal symbols and all grammar rules. Every nonterminal is followed by its so-called FIRST set. This set contains all terminals that can be the first terminal of

Example:

```
Nonterminals and Rules

0_intern, FIRST: {IDENTIFIER '{' }
   1 compound_statement _EOF_
   2 declaration

compound_statement, FIRST: {'{' }
   3 '{' declaration_list statement_list '}'

declaration_list, FIRST: {IDENTIFIER Epsilon }
   4
   5 declaration_list declaration

statement_list, FIRST: {IDENTIFIER Epsilon }
   6
   7 statement_list statement

declaration, FIRST: {IDENTIFIER }
   8 1_declaration_Trial_1 declaration_specifier declarator ';'

...

1_declaration_Trial_1, FIRST: {Epsilon }
  21
```

all possible derivations from the nonterminal. The special entry *Epsilon* indicates that an empty sequence can be derived from the nonterminal. After the FIRST set follow all right-hand sides or rules of a nonterminal. Every rule is printed on a separate line and begins with a rule number. If a line contains only a number (e. g. 4) then this represents an empty right-hand side. The first nonterminal named 0_intern is always added to a grammar. Its right-hand sides are all possible start symbols of the grammar. In the example below there is the regular start symbol *compound_statement* and the start symbol *declaration* which is needed for trial parsing. Nonterminals introduced by the parser generator because of necessary grammar transformations have names consisting of the number of the right-hand side for this nonterminal, the name of the nonterminal, a kind indicator, and a position number within the right-hand side. The kind indicators are Act, UCAct, Pred, Trial, or Prec and they describe the construct that caused a grammar transformation. In the following example the syntactic predicate ? declaration has to be moved out of rule number 8. It is replaced by the artificial nonterminal 1_declaration_Trial_1 and an empty rule with number 21 is added for this nonterminal. The syntactic predicate is moved from rule 8 to rule 21 which is not shown in the listing.

7.3. Automaton: States and Situations

The option -y selects a readable output of the table that controls the generated parser. It consists out of a list of states. Every state has two numbers: an external and an internal one. The external state number is used in the generated parser and it appears in the trace of parsing steps. The internal state number is used in messages of the parser generator. Every state is characterized by a set of so-called situations or items which induce the possible parsing actions.

The output for a state begins with the two state numbers. This is followed by the actions that are possible in this state. The list of actions contains triples and may range over several lines. The first part of a triple is either a terminal or a nonterminal. The second part abbreviates an action which is specified further by the number that constitutes the third part of the triple:

s	n	stands for	shift a token and set the state to n
r	n	stands for	reduce rule number n
sr	n	stands for	shift a token and reduce rule number n
d	n	stands for	dynamic decision scheme n

The action sr = *shift reduce* is a combination of a shift action and a reduce action. Actions of this kind may result from the elimination of LR(0) reductions which is performed by default and which can be disabled with option -r.

The meaning of a triple is: If the parser is in this state and the lookahead token is equal to the first part of a triple then the action specified by the rest of the triple is to be executed. The parser uses nonterminals as "lookahead" symbols, too. This is the case after reduce actions. Like a shift action, a reduce action changes the parser state, too. The details of the parsing algorithm of *Lark* are the same as for *Lalr* and can be found in [Gro90]. A dynamic decision scheme is executed if a predicate is used to solve an LR conflict during run time of the parser. In general, several predicates may be checked where some of the predicates may trigger a trial parse until a decision is made whether to perform a shift action or a reduce action for a certain rule.

After the list of actions follows a set of situations or items that characterize a state. Every situation is printed on a separate line and it begins with a number. A situation is a grammar rule with a dot in its right-hand side. The dot indicates how far parsing has proceeded when the parser is in this state. The fact that there are several situations to a state can be interpreted as an analysis of several rules at the same time or in parallel. Sometimes, situations are followed by a set of terminal symbols enclosed in curly brackets ’{’ and ’}’. These sets are so-called lookahead sets which are computed only where necessary. The meaning is that if parsing continues from this state and the rule of a situation will finally be recognized then the rule can only be followed by terminals out of its lookahead set.

The first states are the so-called start states of a parser. In the following example there are two start states - state 1 for standard parsing and state 2 for trial parsing.

Example:

```
Automaton (States and Situations)

1, intern: 1, Actions:
    '{' s 3, compound_statement s 4,
   1 0_intern: .compound_statement _EOF_
   2 compound_statement: .'{' declaration_list statement_list '}'

2, intern: 2, Actions:
    IDENTIFIER r 21, declaration sr 2, 1_declaration_Trial_1 s 5,
   3 0_intern: .declaration
   4 declaration: .1_declaration_Trial_1 declaration_specifier declarator ';'
   5 1_declaration_Trial_1: .

3, intern: 3, Actions:
    IDENTIFIER r 4, '}' r 4, declaration_list s 6,
   6 compound_statement:  '{'.declaration_list statement_list '}'
   7 declaration_list: .
   8 declaration_list: .declaration_list declaration

4, intern: 4, Actions:
    _EOF_ r 1,
   9 0_intern:  compound_statement._EOF_

5, intern: 6, Actions:
    IDENTIFIER s 7, declaration_specifier s 8, TYPEDEFname sr 9,
  11 declaration:  1_declaration_Trial_1.declaration_specifier declarator ';'
                   {_EOF_ IDENTIFIER '(' ')' '++' ';' '[' ']' '{' '}' }
  12 declaration_specifier: .TYPEDEFname
  13 TYPEDEFname: .IDENTIFIER
  14 TYPEDEFname: .IDENTIFIER

...

7, intern: 8, Actions:
    IDENTIFIER d 2, '(' d 2, '{' d 2,
  21 TYPEDEFname:  IDENTIFIER. {IDENTIFIER '(' '{' }
  22 TYPEDEFname:  IDENTIFIER. {IDENTIFIER '(' '{' }

...

26, intern: 46, Actions:
    ')' sr 16, '++' sr 15,
 124 expression:  TYPEDEFname '(' expression.')'  {')' '++' }
 125 expression:  expression.'++'  {')' '++' }
```

8. Interfaces

A generated parser has three interfaces: The interface of the parser module itself makes the parse procedure available for e. g. a main program. The parser uses a scanner module whose task is to provide a stream of tokens. In case of syntax errors a few procedures of a module named Errors are necessary to handle error messages. Figure 2 gives an overview of the modules and their interface objects. Circles denote procedures, squares denote variables, and arrows represent procedure calls or variable access. The details of the interfaces depend on the implementation language. They are discussed in language specific sections.

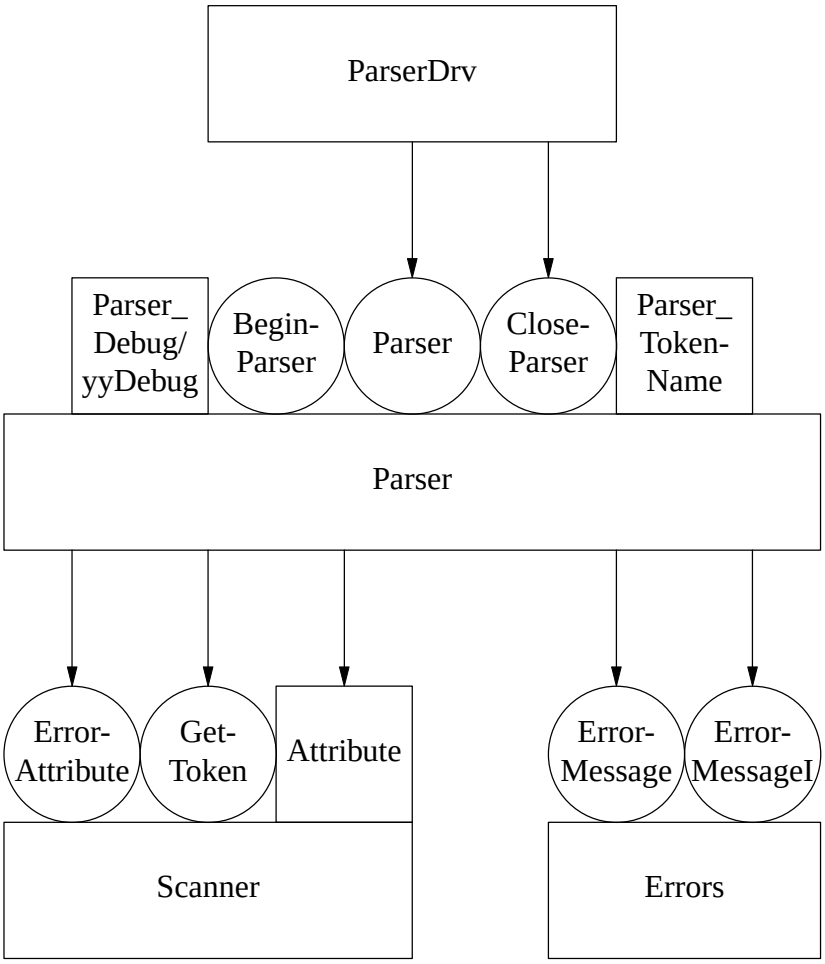


Fig. 2: Interface Objects of the Modules

8.1. C

The option -c selects the generation of a parser in C that can be translated by compilers for ANSI-C, K&R-C, or C++. This is accomplished by appropriate preprocessor directives.

8.1.1. Parser Interface

The parser interface consists of two parts: While the objects specified in the external interface can be used from outside the parser, the objects of the internal interface can be used only within a language description. The external parser interface in the file <Parser>.h has the following contents:

```
# define        yy<start_symbol_1> 1
# define        yy<start_symbol_2> 2
...
# define        t__EOF_            0
# define        t_<token_1>        1
# define        t_<token_2>        2
...
extern  rbool   <Parser>_Debug     ;
extern  char *  <Parser>_TokenName [];
extern  void    Begin<Parser>      (void);
extern  int     <Parser>           (void);
extern  int     <Parser>2          (int StartSymbol);
extern  void    Reset<Parser>      (void);
extern  void    Close<Parser>      (void);
```

- The procedures <Parser> and <Parser>2 are the generated parsing routines. The argument StartSymbol of <Parser>2 specifies the start symbol to be used for parsing. Its value has to be one of the named constants yy<start_symbol_i>. The procedure <Parser> uses as start symbol the start symbol defined first in the grammar or the first nonterminal if no start symbols have been defined. The procedures return the number of syntax errors. A return value of zero indicates a syntactically correct input.
- For every nonterminal that is specified as start symbol a named constant is defined whose name consists of the name of the nonterminal with the prefix 'yy'. These constants are legal arguments for the procedure <Parser>2.
- For every terminal in the grammar a named constant is defined if option -f*prefix* is given. The names for the constants consist of a prefix and the name of the terminal if this is possible. The prefix defaults to 't_'.
- The procedure Reset<Parser> should be called after parsing whenever the execution of the procedures <Parser> or <Parser>2 has been terminated abnormally, e. g. using longjump. This procedure frees any memory allocated by the parser.
- The contents of the target code section named BEGIN is put into a procedure called Begin<Parser>. This procedure is called automatically upon every invocation of the procedure <Parser>. It can also be called explicitly by the user.
- The contents of the target code section named CLOSE is put into a procedure called Close<Parser>. It has to be called explicitly by the user when necessary.
- The variable <Parser>_Debug controls the trace of parsing actions. This trace is useful for debugging a parser. A trace is printed if the parser module is compiled with the option -DYYDEBUG and the variable <Parser>_Debug is set to true (= 1).
- The array <Parser>_TokenName provides a mapping from the internal representation of tokens to the external representation as given in the grammar specification. It maps integers to strings and it is used for example by the parser to provide expressive messages for the error handling module.

The internal parser interface consists of the following objects:

```
# define        yyInitStackSize  100
# define        yyInitBufferSize 100
# define        YYACCEPT
# define        YYABORT

yytNonterminal  yyNonterminal   ;
static  FILE *  yyTrace         = stdout;
static  void    SemActions      (rbool Actions);
static  void    ErrorMessages   (rbool Messages);
static  void    MemoryClear     (int Position);
static  int     GetLookahead    (int k);
static  void    GetAttribute    (int k, <Scanner>_tScanAttribute * Attribute);
static  long    BufferOn        (rbool Actions, rbool Messages);
static  long    BufferOff       (void);
static  long    BufferPosition  ;
static  int     ReParse         (int StartSymbol, int From, int To,
                                 rbool Actions, rbool Messages);
static  void    BufferClear     (void);

# define        TOKENOP
# define        BEFORE_TRIAL
# define        AFTER_TRIAL
# define        ERROR
# define        NO_RECOVER
```

Some of the objects are implemented differently using macros instead of functions.

- The initial size of the parser stack is defined by the value of the preprocessor symbol yyInitStackSize with a default of 100. The stack size is increased automatically when necessary. The initial stack size can be changed by including a preprocessor directive in the GLOBAL section such as:

  ```
  # define yyInitStackSize 200
  ```

- The initial size of the token buffer for trial parsing and reparsing is defined by the value of the preprocessor symbol yyInitBufferSize with a default of 100. The buffer size is increased automatically when necessary. The initial buffer size can be changed by including a preprocessor directive in the GLOBAL section such as:

  ```
  # define yyInitBufferSize 50
  ```

- The statement YYACCEPT can be used in semantic actions. It terminates the current invocation of the internal parsing procedure with the current count of errors.
- The statement YYABORT can be used in semantic actions. It terminates the current invocation of the internal parsing procedure with the current count of errors increased by one.
- The variable yyNonterminal holds the nonterminal of the left-hand side of the rule which has been reduced last. This variable can be accessed in semantic predicates or in semantic actions and this may be of good use in determining whether a trial parse needs to be done. The variable takes on values of enumerated constants generated for every nonterminal whose names are constructed by prefixing the name of the nonterminal with *yyNT*. Example:

  ```
  yyNTdeclaration
  yyNTstatement
  yyNTexpression
  ```

- The variable yyTrace selects the file used for output of the trace information. The default value is stdout. This value can be changed by assignments in the BEGIN section or in semantic actions.
- The procedure SemActions controls the execution of conditional semantic actions without a selection mask (see section 6.3.). The values of the argument Actions mean: true = enable, false = disable. The selection remains valid until a new call of the procedure or the termination of the current invocation of the internal parsing procedure.
- The procedure ErrorMessages controls the reporting of error messages by the parser. The values of the argument Messages mean: true = enable, false = disable. The selection remains valid until a new call of the procedure or the termination of the current invocation of the internal parsing procedure.
- The procedure MemoryClear is useful when the option -b is set to instruct the generated parser to memorize and reuse the results of previous trial parses. These results are stored at the positions in the token buffer where trial parses started. A call of MemoryClear clears a memorized result at the position in the token buffer given by the argument Position. The values for the position can be obtained from the variable BufferPosition (see below).
- The function GetLookahead returns the k-th lookahead token.
- The procedure GetAttribute returns via its second argument called Attribute the additional properties or attributes of the k-th lookahead token.
- The function BufferOn switches the parser to the mode buffer. This is only possible if the parser is in the mode standard, otherwise the mode remains unchanged. The arguments Actions and Messages control the execution of semantic actions and the reporting of error messages in the same way as the procedures SemActions and ErrorMessages. The function returns the position of the current lookahead token in the token buffer.
- The function BufferOff switches the parser to the mode standard. This is only possible if the parser is in the mode buffer, otherwise the mode remains unchanged. The selection for Actions and Messages is reset to the state before the call of BufferOn. The function returns the position of the current lookahead token in the token buffer.
- The variable BufferPosition holds the position of the current lookahead token in the token buffer. Its value can be read at any time while parsing in the mode buffer in order to request positions of arbitrary tokens in the token buffer.
- The function ReParse initiates parsing in the mode reparse using the start symbol StartSymbol. The contents of the token buffer from (including) buffer position From up to (excluding) buffer position To is parsed again according to the start symbol. The arguments Actions and Messages control the execution of semantic actions and the reporting of error messages in the same way as the procedures SemActions and ErrorMessages. The function returns the number of syntax errors found during its invocation.
- The procedure BufferClear signals that the contents of the token buffer can be deleted. The tokens will not be deleted immediately but when it is safe to do this. It is important to tell the parser when buffered tokens can be deleted, because otherwise all remaining input tokens will be buffered. Therefore every call of BufferOn should be accompanied by a call of BufferClear.
- The macro TOKENOP can be used to execute a piece of user code just before every token is consumed by the parser. This feature works only if the parser uses the token buffer. This is the case if trial parsing is activated or one of the following routines is used: GetLookahead, GetAttribute, BufferOn, BufferOff, ReParse, or BufferClear. Then the macro is executed every time a token is

consumed either the first time coming from the source or repeatedly coming from the token buffer. The macro is located before the return statement of a function that provides the parser with tokens. The current token is stored in the local variable yyToken:

```
TOKENOP
return yyToken;
```

- The macro BEFORE_TRIAL can be used to execute statements before the start of every trial parsing.
- The macro AFTER_TRIAL can be used to execute statements after the end of every trial parsing. For example one application is to undo side-effects of trial parsing that fails.
- The macro ERROR can be used to execute statements in case of a syntax error before the execution of the builtin error recovery. Example:

```
  # define ERROR fprintf (stderr, "syntax error at %d, %d\n", \
          Attribute.Position.Line, Attribute.Position.Column);
```

- If the preprocessor symbol NO_RECOVER is defined then the builtin error recovery is excluded from the generated parser. The parser will terminate upon the first syntax error.

8.1.2. Scanner Interface

The generated parser needs some objects usually provided by a scanner module. This module should have a header file called <Scanner>.h to satisfy the include directive of the parser. This header file has to provide the following objects:

```
# include "Position.h"
typedef struct { tPosition Position; } <Scanner>_tScanAttribute;
extern  void <Scanner>_ErrorAttribute (int Token,
                                       <Scanner>_tScanAttribute * Attribute);
extern  <Scanner>_tScanAttribute <Scanner>_Attribute;
extern  int <Scanner>_GetToken (void);
```

- The procedure <Scanner>_GetToken is repeatedly called by the parser in order to receive a stream of tokens. Every call has to return the internal integer representation of the "next" token. The end of the input stream (end of file) is indicated by a value of zero. The integer values returned by the procedure <Scanner>_GetToken have to lie in a range between zero and the maximal value defined in the grammar. This condition is not checked by the parser and values outside of this range may lead to undefined behaviour.
- Additional properties or attributes of tokens are communicated from the scanner to the parser via the global variable <Scanner>_Attribute. For tokens with additional properties like e. g. numbers or identifiers, the procedure <Scanner>_GetToken has to supply the value of this variable as side-effect. The type of this variable can be chosen freely as long as it is an extension of the record type <Scanner>_tScanAttribute.
- The variable <Scanner>_Attribute must have a field called Position which describes the source coordinates of the current token. It has to be computed as side-effect of the procedure <Scanner>_GetToken. In case of syntax errors, this field is passed as argument to the error handling routines.
- The type tParsAttribute must be a record type with at least one field called Scan of type <Scanner>_tScanAttribute. Additional properties or attributes of tokens are transferred from the global

variable <Scanner>_Attribute to this field. See section 2.9 for an example.

- During automatic error repair, a parser may insert tokens. In this case the parser calls the procedure <Scanner>_ErrorAttribute in order to obtain the additional properties or attributes of an inserted token which is given by the argument Token. The procedure should return in the second argument called Attribute default values for the additional properties of this token.

  Example:

```
void ErrorAttribute (int Token, tScanAttribute * pAttribute)
{
   pAttribute->Position = Attribute.Position;
   switch (Token) {
      case /* Ident   */ 1: pAttribute->Ident   = NoIdent; break;
      case /* Integer */ 2: pAttribute->Integer = 0      ; break;
      case /* Real    */ 3: pAttribute->Real    = 0.0    ; break;
   }
}
```

8.1.3. Error Interface

In case of syntax errors, the generated parser calls procedures in order to provide information about the position of the error, the set of expected tokens, and the behaviour of the repair and recovery mechanism. These procedures are conveniently implemented in a separate error handling module. The information provided by the parser may be stored or processed in an arbitrary way. A prototype error handling module is contained in the library *Reuse* (files reuse/c/Errors.h and reuse/c/Errors.c) whose procedures immediately print the information passed as arguments. This module should have a header file called Errors.h to satisfy the include directive in the parser. The header file has to provide the following objects:

```
# define xxSyntaxError          1       /* error codes          */
# define xxExpectedTokens       2
# define xxRestartPoint         3
# define xxTokenInserted        4
# define xxTokenFound           6

# define xxError                3       /* error classes        */
# define xxRepair               5
# define xxInformation          7

# define xxString               7       /* info classes         */

extern void ErrorMessage  (int ErrorCode, int ErrorClass, tPosition Position);
extern void ErrorMessageI (int ErrorCode, int ErrorClass, tPosition Position,
                           int InfoClass, char * Info);
extern void Message       (char * ErrorText, int ErrorClass, tPosition Position);
```

- There are five kinds of messages a generated parser may report. They are encoded by the first group of the above constant definitions. The messages are classified with respect to severity according to the second group of constant definitions.
- The procedure ErrorMessage is used by the parser to report a message, its class, and its source position. It is used for syntax errors and recovery locations (restart points).
- The procedure ErrorMessageI is like the procedure ErrorMessage with additional Information. The latter is characterized by a class or type indication (InfoClass) and an untyped pointer (Info).

During error repair, tokens might be inserted. These are reported one by one and they are classified as xxString (char *). At every syntax error the set of legal or expected tokens is reported using the classification xxString, too.

- The procedure Message is similar to the procedure ErrorMessage. The first argument specifies a message text instead of an error code. This procedure might be called only if the reparsing feature is used.

8.1.4. Parser Driver

A main program is necessary for the test of a generated parser. The parser generator can provide a minimal main program in the file <Parser>Drv.c which can serve as test driver. It has the following contents:

```
# include "<Parser>.h"

int main (void)
{
   (void) <Parser> ();
   Close<Parser>   ();
   return 0;
}
```

8.2. C++

The option -+ selects the generation of a parser in the language C++.

8.2.1. Parser Interface

The parser interface consists of two parts: While the objects specified in the external interface can be used from outside the parser, the objects of the internal interface can be used only within a language description. The external parser interface in the file <Parser>.h defines a class named <Parser> and it has the following contents:

```
# define rbool char

# define <Parser>_BASE_CLASS

class <Parser> <Parser>_BASE_CLASS {

public:
# define               yy<start_symbol_1> 1
# define               yy<start_symbol_2> 2
...

# define               t__EOF_            0
# define               t_<token_1>        1
# define               t_<token_2>        2
...
   const char * const * TokenName       ;
   rbool                yyDebug         ;
                        <Parser>        (void);
                        <Parser>        (<Scanner> *, Errors * = & gErrors);
   int                  Parse           (void);
   int                  Parse           (int StartSymbol);
   void                 Reset           (void);
                        ~<Parser>       (void);
   <Scanner> *          ScannerObj      ;
   Errors *             ErrorsObj       ;
};
```

- The overloaded method Parse is the generated parsing procedure. The argument StartSymbol specifies the start symbol to be used for parsing. Its value has to be one of the named constants yy<start_symbol_i>. Without argument, the start symbol defined first in the grammar or the first nonterminal if no start symbols have been defined are used as start symbol. The method returns the number of syntax errors. A return value of zero indicates a syntactically correct input.
- For every nonterminal that is specified as start symbol a named constant is defined whose name consists of the name of the nonterminal with the prefix 'yy'. These constants are legal arguments for the method Parse.
- For every terminal in the grammar a named constant is defined if option -f*prefix* is given. The names for the constants consist of a prefix and the name of the terminal if this is possible. The prefix defaults to 't_'.
- The method Reset should be called after parsing whenever the execution of the method Parse has been terminated abnormally, e. g. using exception handling. This method frees any memory allocated by the parser.
- The member ScannerObj refers to the scanner used by the parser. This has to be an object from the class <Scanner>. This object has to provide the features described below in the section "Scanner Interface". A scanner object has to be created and assigned to the member ScannerObj before the method Parse is called. See the section "Parser Driver" for an example.
- The member ErrorsObj refers to the error handler used by the parser. This has to be an object from the class Errors. This object has to provide the features described below in the section "Error Interface". The member ErrorsObj initially refers to the object gErrors from the library *Reuse* (files reuse/cpp/Global.h and reuse/cpp/Global.c).
- The contents of the target code section named BEGIN is put into a constructor called <Parser>. It is called automatically upon initialization of a parser object.

- There is another constructor called <Parser> available which has two arguments. It does the same as the constructor without arguments. Additionally, the members ScannerObj and ErrorsObj are initialized with the values given as arguments. The second argument is optional. If not given the default object gErrors is used.
- The contents of the target code section named CLOSE is put into a destructor called ~<Parser>. It is called automatically upon finalization of a parser object.
- The member variable yyDebug controls the trace of parsing actions. This trace is useful for debugging a parser. A trace is printed if the parser module is compiled with the option -DYYDEBUG and the variable yyDebug is set to true (= 1).
- The "array" TokenName provides a mapping from the internal representation of tokens to the external representation as given in the grammar specification. It maps integers to strings and it is used for example by the parser to provide expressive messages for the error handling module.
- The preprocessor symbol <Parser>_BASE_CLASS can be used to specify a base class for the class <Parser> using a #define directive in the EXPORT section of a language description. Example:

```
EXPORT {
# define Parser_BASE_CLASS : public BaseClass
}
```

The internal parser interface consists of the following objects:

```
# define        yyInitStackSize  100
# define        yyInitBufferSize 100
# define        YYACCEPT
# define        YYABORT

yytNonterminal  yyNonterminal   ;
static  FILE *  yyTrace         = stdout;
static  void    SemActions      (rbool Actions);
static  void    ErrorMessages   (rbool Messages);
static  void    MemoryClear     (int Position);
static  int     GetLookahead    (int k);
static  void    GetAttribute    (int k, <Scanner>_tScanAttribute * Attribute);
static  long    BufferOn        (rbool Actions, rbool Messages);
static  long    BufferOff       (void);
static  long    BufferPosition  ;
static  int     ReParse         (int StartSymbol, int From, int To,
                                 rbool Actions, rbool Messages);
static  void    BufferClear     (void);

# define        TOKENOP
# define        BEFORE_TRIAL
# define        AFTER_TRIAL
# define        ERROR
# define        NO_RECOVER
```

Some of the objects are implemented differently using macros instead of functions.

- The initial size of the parser stack is defined by the value of the preprocessor symbol yyInitStackSize with a default of 100. The stack size is increased automatically when necessary. The initial stack size can be changed by including a preprocessor directive in the GLOBAL section such as:

```
# define yyInitStackSize 200
```

- The initial size of the token buffer for trial parsing and reparsing is defined by the value of the preprocessor symbol yyInitBufferSize with a default of 100. The buffer size is increased automatically when necessary. The initial buffer size can be changed by including a preprocessor directive in the GLOBAL section such as:

```
# define yyInitBufferSize 50
```

- The statement YYACCEPT can be used in semantic actions. It terminates the current invocation of the internal parsing procedure with the current count of errors.
- The statement YYABORT can be used in semantic actions. It terminates the current invocation of the internal parsing procedure with the current count of errors increased by one.
- The variable yyNonterminal holds the nonterminal of the left-hand side of the rule which has been reduced last. This variable can be accessed in semantic predicates or in semantic actions and this may be of good use in determining whether a trial parse needs to be done. The variable takes on values of enumerated constants generated for every nonterminal whose names are constructed by prefixing the name of the nonterminal with *yyNT*. Example:

```
yyNTdeclaration
yyNTstatement
yyNTexpression
```

- The variable yyTrace selects the file used for output of the trace information. The default value is stdout. This value can be changed by assignments in the BEGIN section or in semantic actions.
- The procedure SemActions controls the execution of conditional semantic actions without a selection mask (see section 6.3.). The values of the argument Actions mean: true = enable, false = disable. The selection remains valid until a new call of the procedure or the termination of the current invocation of the internal parsing procedure.
- The procedure ErrorMessages controls the reporting of error messages by the parser. The values of the argument Messages mean: true = enable, false = disable. The selection remains valid until a new call of the procedure or the termination of the current invocation of the internal parsing procedure.
- The procedure MemoryClear is useful when the option -b is set to instruct the generated parser to memorize and reuse the results of previous trial parses. These results are stored at the positions in the token buffer where trial parses started. A call of MemoryClear clears a memorized result at the position in the token buffer given by the argument Position. The values for the position can be obtained from the variable BufferPosition (see below).
- The function GetLookahead returns the k-th lookahead token.
- The procedure GetAttribute returns via its second argument called Attribute the additional properties or attributes of the k-th lookahead token.
- The function BufferOn switches the parser to the mode buffer. This is only possible if the parser is in the mode standard, otherwise the mode remains unchanged. The arguments Actions and Messages control the execution of semantic actions and the reporting of error messages in the same way as the procedures SemActions and ErrorMessages. The function returns the position of the current lookahead token in the token buffer.

- The function BufferOff switches the parser to the mode standard. This is only possible if the parser is in the mode buffer, otherwise the mode remains unchanged. The selection for Actions and Messages is reset to the state before the call of BufferOn. The function returns the position of the current lookahead token in the token buffer.
- The variable BufferPosition holds the position of the current lookahead token in the token buffer. Its value can be read at any time while parsing in the mode buffer in order to request positions of arbitrary tokens in the token buffer.
- The function ReParse initiates parsing in the mode reparse using the start symbol StartSymbol. The contents of the token buffer from (including) buffer position From up to (excluding) buffer position To is parsed again according to the start symbol. The arguments Actions and Messages control the execution of semantic actions and the reporting of error messages in the same way as the procedures SemActions and ErrorMessages. The function returns the number of syntax errors found during its invocation.
- The procedure BufferClear signals that the contents of the token buffer can be deleted. The tokens will not be deleted immediately but when it is safe to do this. It is important to tell the parser when buffered tokens can be deleted, because otherwise all remaining input tokens will be buffered. Therefore every call of BufferOn should be accompanied by a call of BufferClear.
- The macro TOKENOP can be used to execute a piece of user code just before every token is consumed by the parser. This feature works only if the parser uses the token buffer. This is the case if trial parsing is activated or one of the following routines is used: GetLookahead, GetAttribute, BufferOn, BufferOff, ReParse, or BufferClear. Then the macro is executed every time a token is consumed either the first time coming from the source or repeatedly coming from the token buffer. The macro is located before the return statement of a function that provides the parser with tokens. The current token is stored in the local variable yyToken:

```
TOKENOP
return yyToken;
```

- The macro BEFORE_TRIAL can be used to execute statements before the start of every trial parsing.
- The macro AFTER_TRIAL can be used to execute statements after the end of every trial parsing. For example one application is to undo side-effects of trial parsing that fails.
- The macro ERROR can be used to execute statements in case of a syntax error before the execution of the builtin error recovery. Example:

```
# define ERROR fprintf (stderr, "syntax error at %d, %d\n", \
        Attribute.Position.Line, Attribute.Position.Column);
```

- If the preprocessor symbol NO_RECOVER is defined then the builtin error recovery is excluded from the generated parser. The parser will terminate upon the first syntax error.

8.2.2. Scanner Interface

The generated parser needs some features usually provided by a scanner class. This class should have a header file called <Scanner>.h to satisfy the include directive of the parser. This header file has to define a class named <Scanner> which provides the following objects:

```
# include "Position.h"

typedef struct { tPosition Position; } <Scanner>_tScanAttribute;

class <Scanner> {

public:
   <Scanner>_tScanAttribute Attribute   ;
   int                      GetToken    (void);
   void                     ErrorAttribute  (int Token,
                                             <Scanner>_tScanAttribute * Attribute);
};
```

- The method GetToken is repeatedly called by the parser in order to receive a stream of tokens. Every call has to return the internal integer representation of the "next" token. The end of the input stream (end of file) is indicated by a value of zero. The integer values returned by the method GetToken have to lie in a range between zero and the maximal value defined in the grammar. This condition is not checked by the parser and values outside of this range may lead to undefined behaviour.
- Additional properties or attributes of tokens are communicated from the scanner to the parser via the member Attribute. For tokens with additional properties like e. g. numbers or identifiers, the method GetToken has to supply the value of this variable as side-effect. The type of this variable can be chosen freely as long as it is an extension of the record type <Scanner>_tScanAttribute.
- The member Attribute must have a field called Position which describes the source coordinates of the current token. It has to be computed as side-effect of the method GetToken. In case of syntax errors, this field is passed as argument to the error handling routines.
- The type <Parser>_tParsAttribute must be a record type with at least one field called Scan of type <Scanner>_tScanAttribute. Additional properties or attributes of tokens are transferred from the member Attribute to this field. See section 2.9 for an example.
- During automatic error repair, a parser may insert tokens. In this case the parser calls the method ErrorAttribute in order to obtain the additional properties or attributes of an inserted token which is given by the argument Token. The procedure should return in the second argument called Attribute default values for the additional properties of this token.

  Example:

```
void Scanner::ErrorAttribute (int Token, tScanAttribute * pAttribute)
{
   pAttribute->Position = Attribute.Position;
   switch (Token) {
      case /* Ident   */ 1: pAttribute->Ident   = NoIdent; break;
      case /* Integer */ 2: pAttribute->Integer = 0      ; break;
      case /* Real    */ 3: pAttribute->Real    = 0.0    ; break;
   }
}
```

8.2.3. Error Interface

In case of syntax errors, the generated parser calls methods in order to provide information about the position of the error, the set of expected tokens, and the behaviour of the repair and recovery mechanism. These methods are conveniently implemented in a separate error handling class. The information provided by the parser may be stored or processed in an arbitrary way. A prototype error handling class is contained in the library *Reuse* (files reuse/cpp/Errors.h and

reuse/cpp/Errors.cxx) whose procedures immediately print the information passed as arguments. This module should have a header file called Errors.h to satisfy the include directive in the parser. The header file has to provide the following objects:

```
const   xxSyntaxError           = 1;    /* error codes          */
const   xxExpectedTokens        = 2;
const   xxRestartPoint          = 3;
const   xxTokenInserted         = 4;
const   xxTokenFound            = 6;

const   xxError                 = 3;    /* error classes        */
const   xxRepair                = 5;
const   xxInformation           = 7;

const   xxString                = 7;    /* info classes         */

class Errors {
 public:
  void ErrorMessage  (int ErrorCode, int ErrorClass, tPosition Position);
  void ErrorMessageI (int ErrorCode, int ErrorClass, tPosition Position,
                      int InfoClass, char * Info);
  void Message       (char * ErrorText, int ErrorClass, tPosition Position);
};
```

- There are five kinds of messages a generated parser may report. They are encoded by the first group of the above constant definitions. The messages are classified with respect to severity according to the second group of constant definitions.
- The procedure ErrorMessage is used by the parser to report a message, its class, and its source position. It is used for syntax errors and recovery locations (restart points).
- The procedure ErrorMessageI is like the procedure ErrorMessage with additional Information. The latter is characterized by a class or type indication (InfoClass) and an untyped pointer (Info). During error repair, tokens might be inserted. These are reported one by one and they are classified as xxString (char *). At every syntax error the set of legal or expected tokens is reported using the classification xxString, too.
- The procedure Message is similar to the procedure ErrorMessage. The first argument specifies a message text instead of an error code. This procedure might be called only if the reparsing feature is used.

8.2.4. Parser Driver

A main program is necessary for the test of a generated parser. The parser generator can provide a minimal main program in the file <Parser>Drv.cxx which can serve as test driver. It has the following contents:

```
# include "<Scanner>.h"
# include "<Parser>.h"

int main (void)
{
   <Parser> ParserObj;
   ParserObj.ScannerObj = new <Scanner>;
   return ParserObj.Parse ();
}
```

8.3. Java

The option -j selects the generation of a parser in the language Java.

8.3.1. Parser Interface

The parser interface consists of two parts: While the objects specified in the external interface can be used from outside the parser, the objects of the internal interface can be used only within a language description. The external parser interface consists of the public classes and methods in the file <Parser>.java which defines at least a class named <Parser>. Additional classes may be defined in the GLOBAL and EXPORT sections: those in EXPORT become inner classes of <Parser> while those in GLOBAL are separate from <Parser> but share the same source file. Be aware that declaring more than one class in a file in this way is not usual practice and may confuse some Java tools, so classes should normally be declared in the EXPORT section.

The class <Parser> has the following public methods and fields:

```
class <Parser> {
   /* start symbols from START section if specified */
   static final int yy<start_symbol_1> 1
   static final int yy<start_symbol_2> 2
   ...
   static final int t__EOF_            0
   static final int t_<token_1>        1
   static final int t_<token_2>        2
   ...
   public String        yyGetTokenName  (int token);
   public boolean       yyDebug         = false;
   public               <Parser>        (<Scanner> scanner)
                                           throws java.io.IOException;
   public int           parse           () throws java.io.IOException;
   public int           parse           (int yyStartSymbol)
                                           throws java.io.IOException;
   public <Scanner>     scanner         ;
   public Errors        errorsObj       = Global.errors;
}
```

- The overloaded method *parse* is the generated parsing procedure. The argument *yyStartSymbol* specifies the start symbol to be used for parsing. Its value has to be one of the named constants yy<start_symbol_i>. Without argument, the start symbol defined first in the grammar or the first nonterminal if no start symbols have been defined is used as start symbol. The method returns the number of syntax errors. A return value of zero indicates a syntactically correct input.
- For every nonterminal that is specified as start symbol a named constant is defined whose name consists of the name of the nonterminal with the prefix 'yy'. These constants are legal arguments for the method *parse*.
- For every terminal in the grammar a named constant is defined if option -f*prefix* is given. The names for the constants consist of a prefix and the name of the terminal if this is possible. The prefix defaults to 't_'.
- The member *scanner* refers to the scanner used by the parser. This has to be an object from the class <Scanner>. This object has to provide the features described below in the section "Scanner Interface". A scanner object has to be created and passed to the <Parser> constructor. See the

section "Parser Driver" for an example.

- The member *errorsObj* refers to the error handler used by the parser. This has to be an object from the class Errors. This object has to provide the features described below in the section "Error Interface". The member *errorsObj* initially refers to the object *Global.errors* from the library *Reuse* (class Global).
- The contents of the target code section named BEGIN is put into the <Parser> constructor and is executed automatically upon initialization of a parser object.
- The contents of the target code section named CLOSE is put into a method called *finalize*. This method is usually called automatically by the JVM before the <Parser> object is garbage collected, or it may be called directly.
- The field *yyDebug* controls the trace of parsing actions. This trace is useful for debugging a parser. A trace is printed if the C preprocessor macro YYDEBUG is defined in the GLOBAL section and *yyDebug* is set to true.
- The method *yyGetTokenName* provides a mapping from the internal representation of tokens to the external representation as given in the grammar specification. It maps integers to strings and it is used for example by the parser to provide expressive messages for the error handling module.

The internal parser interface consists of private fields and methods from the <Parser> class with additional features implemented using the C preprocessor language:

```
# define             yyScanAttribute  <Scanner>.ScanAttribute
# define             yyInitStackSize  100
# define             yyInitBufferSize 100
# define             YYACCEPT
# define             YYABORT

/* local */ int      yyNonterminal;
private CocktailWriter yyTrace;
private void         semActions      (boolean actions);
private void         errorMessages   (boolean messages);
private void         memoryClear     (int position);
private int          getLookahead    (int k) throws java.io.IOException;
private yyScanAttribute getAttribute (int k)
                                            throws java.io.IOException;
private int          bufferOn        (boolean actions, boolean messages)
                                            throws java.io.IOException;
private int          bufferOff       ();
private int          bufferPosition;
private int          reParse         (
     int yyStartSymbol,
     int yyFrom, int yyTo,
     boolean yyActions, boolean yyMessages
                                     ) throws java.io.IOException;
private void         bufferClear     ();

# define             TOKENOP
# define             BEFORE_TRIAL
# define             AFTER_TRIAL
# define             ERROR
# define             NO_RECOVER
```

Some of the objects are implemented differently using macros instead of functions. The macro

yyScanAttribute is described in Scanner Interface below.

- The initial size of the parser stack is defined by the value of the preprocessor symbol yyInitStackSize with a default of 100. The stack size is increased automatically when necessary. The initial stack size can be changed by including a preprocessor directive in the GLOBAL section such as:

```
# define yyInitStackSize 200
```

- The initial size of the token buffer for trial parsing and reparsing is defined by the value of the preprocessor symbol yyInitBufferSize with a default of 100. The buffer size is increased automatically when necessary. The initial buffer size can be changed by including a preprocessor directive in the GLOBAL section such as:

```
# define yyInitBufferSize 50
```

- The statement YYACCEPT can be used in semantic actions. It terminates the current invocation of the internal parsing procedure with the current count of errors.
- The statement YYABORT can be used in semantic actions. It terminates the current invocation of the internal parsing procedure with the current count of errors increased by one.
- The variable *yyNonterminal* holds the nonterminal of the left-hand side of the rule which has been reduced last. This variable can be accessed in semantic predicates or in semantic actions and this may be of good use in determining whether a trial parse needs to be done. The variable takes on values of constants generated for every nonterminal whose names are constructed by prefixing the name of the nonterminal with *yyNT*. Example:

```
yyNTdeclaration
yyNTstatement
yyNTexpression
```

- The variable *yyTrace* selects the file used for output of the trace information. The default value is stdout. This value can be changed by assignments in the BEGIN section or in semantic actions.
- The procedure *semActions* controls the execution of conditional semantic actions without a selection mask (see section 6.3.). The values of the argument *actions* mean: true = enable, false = disable. The selection remains valid until a new call of the procedure or the termination of the current invocation of the internal parsing procedure.
- The procedure *errorMessages* controls the reporting of error messages by the parser. The values of the argument *messages* mean: true = enable, false = disable. The selection remains valid until a new call of the procedure or the termination of the current invocation of the internal parsing procedure.
- The method *memoryClear* is useful when the option -b is set to instruct the generated parser to memorize and reuse the results of previous trial parses. These results are stored at the positions in the token buffer where trial parses started. A call of *memoryClear* clears a memorized result at the position in the token buffer given by the argument Position. The values for the position can be obtained from the field *bufferPosition* (see below).
- The method *getLookahead* returns the k-th lookahead token.
- The method *getAttribute* returns the additional properties or attributes of the k-th lookahead token.

- The method *bufferOn* switches the parser to the mode buffer. This is only possible if the parser is in the mode standard, otherwise the mode remains unchanged. The arguments *actions* and *messages* control the execution of semantic actions and the reporting of error messages in the same way as the methods *semActions* and *errorMessages*. The function returns the position of the current lookahead token in the token buffer.
- The method *bufferOff* switches the parser to the mode standard. This is only possible if the parser is in the mode buffer, otherwise the mode remains unchanged. The selection for Actions and Messages is reset to the state before the call of *bufferOn*. The function returns the position of the current lookahead token in the token buffer.
- The field *bufferPosition* holds the position of the current lookahead token in the token buffer. Its value can be read at any time while parsing in the mode buffer in order to request positions of arbitrary tokens in the token buffer.
- The method *reParse* initiates parsing in the mode reparse using the start symbol *yyStartSymbol*. The contents of the token buffer from (including) buffer position *yyFrom* up to (excluding) buffer position *yyTo* is parsed again according to the start symbol. The arguments *yyActions* and *yyMessages* control the execution of semantic actions and the reporting of error messages in the same way as the methods *semActions* and *errorMessages*. The function returns the number of syntax errors found during its invocation.
- The method *bufferClear* signals that the contents of the token buffer can be deleted. The tokens will not be deleted immediately but when it is safe to do this. It is important to tell the parser when buffered tokens can be deleted, because otherwise all remaining input tokens will be buffered. Therefore every call of *bufferOn* should be accompanied by a call of *bufferClear*.
- The macro TOKENOP can be used to execute a piece of user code just before every token is consumed by the parser. This feature works only if the parser uses the token buffer. This is the case if trial parsing is activated or one of the following routines is used: getLookahead, getAttribute, bufferOn, bufferOff, reParse, or bufferClear. Then the macro is executed every time a token is consumed either the first time coming from the source or repeatedly coming from the token buffer. The macro is located before the return statement of a function that provides the parser with tokens. The current token is stored in the local variable yyToken:

```
TOKENOP
return yyToken;
```

- The macro BEFORE_TRIAL can be used to execute statements before the start of every trial parsing.
- The macro AFTER_TRIAL can be used to execute statements after the end of every trial parsing. For example one application is to undo side-effects of trial parsing that fails.
- The macro ERROR can be used to execute statements in case of a syntax error before the execution of the builtin error recovery. Example:

```
# define ERROR System.err.println (
      "syntax error at " + attribute.position());
```

- If the preprocessor symbol NO_RECOVER is defined then the builtin error recovery is excluded from the generated parser. The parser will terminate upon the first syntax error.

8.3.2. Scanner Interface

The generated parser obtains input tokens from an instance of class <Scanner> which provides the following:

- A method *getToken* which returns the code of the next input token.
- A means of communicating any additional attributes of the current token, for example the value of a numeric constant or a dictionary pointer associated with an identifier. The parser needs to obtain the input position of any token for use in syntax error messages.
- A means of constructing a suitable attribute value for a token inserted by the parser during error repair.

The details of this interface may be tuned by preprocessor macros defined in the GLOBAL section of the scanner and parser specifications. First we describe the default arrangement, and then we will discuss the specification of the macros.

The usual interface to <Scanner> is:

```
class <Scanner> {
   class ScanAttribute implements HasPosition {
      ...
   }

   public ScanAttribute  attribute      ;
   public int            getToken       ();
   public ScanAttribute  errorAttribute (int token);
}
```

- The method *getToken* is repeatedly called by the parser in order to receive a stream of tokens. Every call has to return the internal integer representation of the "next" token. The end of the input stream (end of file) is indicated by a value of zero. The integer values returned by the method *getToken* have to lie in a range between zero and the maximal value defined in the grammar. This condition is not checked by the parser and values outside of this range may lead to undefined behaviour.
- Additional properties or attributes of tokens are communicated from the scanner to the parser via the field *attribute*. For tokens with additional properties like e. g. numbers or identifiers, the method *getToken* has to supply the value of this variable as side-effect. The type of this variable can be chosen freely as long as it implements the *HasPosition* interface from the package de.cocolab.reuse. This provides access to an instance of class Position which describes the source coordinates of the current token. In case of syntax errors, this field is passed as argument to the error handling routines.
- The macro *yyParsAttribute* defines the type of object used for access to attributes from semantic actions. For terminals the attribute is derived from the scanner attribute: by default *yyParsAttribute* is defined to be java.lang.Object and the scanner attribute is copied by assignment.
- During automatic error repair, a parser may insert tokens. In this case the parser calls the method *errorAttribute* in order to obtain the additional properties or attributes of an inserted token which is given by the argument *token*. The procedure should return default values for the additional properties of this token.

Example:

```
public ScanAttribute errorAttribute (int token)
{
   /* Create a new attribute object based on the current attribute, i.e.
    * copying the current position data.
    */
   ScanAttribute result = new ScanAttribute (attribute);

   /* Set the appropriate field according to the token type
    */
   switch (token) {
      case /* Ident   */ 1: result.ident   = NoIdent; break;
      case /* Integer */ 2: result.integer = 0      ; break;
      case /* Real    */ 3: result.real    = 0.0    ; break;
   }
   return result;
}
```

8.3.3. Tailoring the Parser

This section discusses the choices that are available to:

- determine the type associated with the attribute stack, that is the type used to hold S-attributes for non-terminals and terminals;
- determine the type of the attribute obtained from the scanner, that is the type used to access attributes of terminals;
- the relationship between these two types.

The type used for elements on the attribute stack is determined by the macro *yyParsAttribute* which defaults to java.lang.Object. Any object may be used as an attribute but a cast is usually required to access attributes.

This choice yields rules of the form:

```
expr: expr '+' expr  { $$ = new AddExpr ( (Expr)$1, (Expr)$3 ); }
```

A development of this scheme involves building a tree of attribute classes related by inheritance. The root class is used as the attribute type, and has methods used to obtain the correct static type as in the following example.

```
GLOBAL {
  #define yyParsAttribute Tree
}

EXPORT {
  class Tree {
     public Expr asExpr() {
        // throw ClassCastException
     }
     public AddExpr asAddExpr() {
        // throw ClassCastException
     }
     public DivideExpr asDivideExpr() {
        // throw ClassCastException
     }
     ...
  }

  class Expr extends Tree {
     public Expr Expr() {
        return this;
     }
     ...
  }

  class AddExpr extends Expr {
     AddExpr (Expr lhs, Expr rhs) {
        this.lhs = lhs; this.rhs = rhs;
     }
     public AddExpr asAddExpr() { // overrides method from Tree
        return this;
     }
     public Expr lhs, rhs;
  }
  class DivideExpr extends Expr {
     ...
  }
  ...
}

RULE
    expr: expr '+' expr  { $$ = new AddExpr ( $1.asExpr(), $3.asExpr() ); }
```

We have added the root Tree class together with an 'as' method for every possible descendant which is then used in rules instead of a Java type cast. Performance should be better as a polymorphic dispatch is more efficient than a cast.

A third possiblity is to use a basic type such as int for the attribute class. This is applicable in specialised circumstances, as in the integer_calc example supplied with the Cocktail distribution. This involves the use of a few other specialised macros not described here.

Some more macros deal with the type used for attributes obtained from the scanner and the relationship between this and the parser attribute type.

- *yyScanAttribute* defines the scanner attribute type, by default an inner class ScanAttribute of the scanner class.
- *yyScanToPars(s)* is used to store a scanner attribute in a variable of type yyParsAttribute. If yyScanAttribute is not assignment compatible with yyParsAttribute then it may be necessary to

create a new object to encapsulate the value.

- *yyAttributePosition* describes how to obtain a Position object from a scanner attribute; this is used in formulating parser error messages. The default is suitable for any class which implements the HasPosition interface.
- By default the scanner is assumed to provide the attribute for the current token as a public member called *attribute*. Other schemes may be accomodated by defining the macros *yyGetScanAttribute* and *yyPutScanAttribute* which are documented in the skeleton.

8.3.4. Error Interface

In case of syntax errors, the generated parser calls methods in order to provide information about the position of the error, the set of expected tokens, and the behaviour of the repair and recovery mechanism. These methods are conveniently implemented in a separate error handling class. The information provided by the parser may be stored or processed in an arbitrary way. A default error handling class Errors is contained in the library *Reuse* whose methods either immediately print the information passed as arguments or store it for subsequent output sorted by position. This class is fully documented in the HTML documentation supplied with Cocktail.

8.3.5. Parser Driver

A main program is necessary for the test of a generated parser. The parser generator can provide a minimal main program in the file <Parser>Drv.java which can serve as test driver. It has the following contents:

```
import java.io.*;

class <Parser>Drv {

    public static void main (String [] argv) throws java.io.IOException {

        String filename = null;
        <Parser> parser = new <Parser> (new <Scanner> ());

        for (int i = 0; i < argv.length; i ++) {
            if (argv [i].equals ("-D")) parser.yyDebug = true;
            else filename = argv [i];
        }

        if (filename != null)
            parser.scanner.beginFile (new FileInputStream (filename));

        parser.parse ();
        parser.finalize ();
    }
}
```

To obtain a debugging trace of parser actions put "#define YYDEBUG" in the GLOBAL section of the parser specification and run with the command line option -D. By default input is read from stdin; to read from a file give the file name as a command line argument. For example to read from buggy.in and output a trace:

```
java <Parser>Drv -D buggy.in
```

8.4. Modula-2

8.4.1. Parser Interface

The parser interface consists of two parts: While the objects specified in the external interface can be used from outside the parser, the objects of the internal interface can be used only within a language description. The external parser interface in the file <Parser>.md has the following contents:

```
DEFINITION MODULE <Parser>;

CONST     yy<start_symbol_1> = 1;
CONST     yy<start_symbol_2> = 2;
...

CONST     t__EOF_            = 0;
CONST     t_<token_1>        = 1;
CONST     t_<token_2>        = 2;
...

VAR       yyDebug       : BOOLEAN;

PROCEDURE Begin<Parser> ;
PROCEDURE <Parser>      (): CARDINAL;
PROCEDURE <Parser>2     (StartSymbol: SHORTCARD): CARDINAL;
PROCEDURE Close<Parser> ;
PROCEDURE TokenName     (Token: CARDINAL; VAR Name: ARRAY OF CHAR);

END <Parser>.
```

- The procedures <Parser> and <Parser>2 are the generated parsing routines. The argument StartSymbol of <Parser>2 specifies the start symbol to be used for parsing. Its value has to be one of the named constants yy<start_symbol_i>. The procedure <Parser> uses as start symbol the start symbol defined first in the grammar or the first nonterminal if no start symbols have been defined. The procedures return the number of syntax errors. A return value of zero indicates a syntactically correct input.
- For every nonterminal that is specified as start symbol a named constant is defined whose name consists of the name of the nonterminal with the prefix 'yy'. These constants are legal arguments for the procedure <Parser>2.
- For every terminal in the grammar a named constant is defined if option -f*prefix* is given. The names for the constants consist of a prefix and the name of the terminal if this is possible. The prefix defaults to 't_'.
- The contents of the target code section named BEGIN is put into a procedure called Begin<Parser>. This procedure is called automatically upon every invocation of the procedure <Parser>. It can also be called explicitly by the user.
- The contents of the target code section named CLOSE is put into a procedure called Close<Parser>. It has to be called explicitly by the user when necessary.
- The variable yyDebug controls the trace of parsing actions. This trace is useful for debugging a parser. A trace is printed when the preprocessor directive

  ```
  # define YYDEBUG
  ```

  is included in the GLOBAL section and the variable yyDebug is set to TRUE.

- The procedure TokenName provides a mapping from the internal representation of tokens to the external representation as given in the grammar specification. It maps integers to strings and it is used for example by the parser to provide expressive messages for the error handling module.

The internal parser interface consists of the following objects:

```
# define  yyInitStackSize  100
# define  yyInitBufferSize 100
# define  YYACCEPT
# define  YYABORT

VAR       yyNonterminal : yySymbolRange;
VAR       yyTrace       : IO.tFile; (* := IO.StdOutput; *)
VAR       BufferPosition: INTEGER;

PROCEDURE SemActions    (Actions : BOOLEAN);
PROCEDURE ErrorMessages (Messages: BOOLEAN);
PROCEDURE MemoryClear   (Position: INTEGER);
PROCEDURE GetLookahead  (k: INTEGER): INTEGER;
PROCEDURE GetAttribute  (k: INTEGER; VAR Attribute: <Scanner>.tScanAttribute);
PROCEDURE BufferOn      (Actions, Messages: BOOLEAN): INTEGER;
PROCEDURE BufferOff     (): INTEGER;
PROCEDURE ReParse       (StartSymbol: INTEGER; From, To: INTEGER;
                         Actions, Messages: BOOLEAN): INTEGER;
PROCEDURE BufferClear   ();

# define  TOKENOP
# define  BEFORE_TRIAL
# define  AFTER_TRIAL
# define  ERROR
# define  NO_RECOVER
```

Some of the objects are implemented differently using macros instead of functions.

- The initial size of the parser stack is defined by the value of the preprocessor symbol yyInitStackSize with a default of 100. The stack size is increased automatically when necessary. The initial stack size can be changed by including a preprocessor directive in the GLOBAL section such as:

  ```
  # define yyInitStackSize 200
  ```

- The initial size of the token buffer for trial parsing and reparsing is defined by the value of the preprocessor symbol yyInitBufferSize with a default of 100. The buffer size is increased automatically when necessary. The initial buffer size can be changed by including a preprocessor directive in the GLOBAL section such as:

  ```
  # define yyInitBufferSize 50
  ```

- The statement YYACCEPT can be used in semantic actions. It terminates the current invocation of the internal parsing procedure with the current count of errors.
- The statement YYABORT can be used in semantic actions. It terminates the current invocation of the internal parsing procedure with the current count of errors increased by one.
- The variable yyNonterminal holds the nonterminal of the left-hand side of the rule which has been reduced last. This variable can be accessed in semantic predicates or in semantic actions and this may be of good use in determining whether a trial parse needs to be done. The variable

takes on values of enumerated constants generated for every nonterminal whose names are constructed by prefixing the name of the nonterminal with *yyNT*. Example:

```
yyNTdeclaration
yyNTstatement
yyNTexpression
```

- The variable yyTrace selects the file used for output of the trace information. The default value is stdout. This value can be changed by assignments in the BEGIN section or in semantic actions.
- The procedure SemActions controls the execution of conditional semantic actions without a selection mask (see section 6.3.). The values of the argument Actions mean: true = enable, false = disable. The selection remains valid until a new call of the procedure or the termination of the current invocation of the internal parsing procedure.
- The procedure ErrorMessages controls the reporting of error messages by the parser. The values of the argument Messages mean: true = enable, false = disable. The selection remains valid until a new call of the procedure or the termination of the current invocation of the internal parsing procedure.
- The procedure MemoryClear is useful when the option -b is set to instruct the generated parser to memorize and reuse the results of previous trial parses. These results are stored at the positions in the token buffer where trial parses started. A call of MemoryClear clears a memorized result at the position in the token buffer given by the argument Position. The values for the position can be obtained from the variable BufferPosition (see below).
- The function GetLookahead returns the k-th lookahead token.
- The procedure GetAttribute returns via its second argument called Attribute the additional properties or attributes of the k-th lookahead token.
- The function BufferOn switches the parser to the mode buffer. This is only possible if the parser is in the mode standard, otherwise the mode remains unchanged. The arguments Actions and Messages control the execution of semantic actions and the reporting of error messages in the same way as the procedures SemActions and ErrorMessages. The function returns the position of the current lookahead token in the token buffer.
- The function BufferOff switches the parser to the mode standard. This is only possible if the parser is in the mode buffer, otherwise the mode remains unchanged. The selection for Actions and Messages is reset to the state before the call of BufferOn. The function returns the position of the current lookahead token in the token buffer.
- The variable BufferPosition holds the position of the current lookahead token in the token buffer. Its value can be read at any time while parsing in the mode buffer in order to request positions of arbitrary tokens in the token buffer.
- The function ReParse initiates parsing in the mode reparse using the start symbol StartSymbol. The contents of the token buffer from (including) buffer position From up to (excluding) buffer position To is parsed again according to the start symbol. The arguments Actions and Messages control the execution of semantic actions and the reporting of error messages in the same way as the procedures SemActions and ErrorMessages. The function returns the number of syntax errors found during its invocation.
- The procedure BufferClear signals that the contents of the token buffer can be deleted. The tokens will not be deleted immediately but when it is safe to do this. It is important to tell the parser when buffered tokens can be deleted, because otherwise all remaining input tokens will be

buffered. Therefore every call of BufferOn should be accompanied by a call of BufferClear.

- The macro TOKENOP can be used to execute a piece of user code just before every token is consumed by the parser. This feature works only if the parser uses the token buffer. This is the case if trial parsing is activated or one of the following routines is used: GetLookahead, GetAttribute, BufferOn, BufferOff, ReParse, or BufferClear. Then the macro is executed every time a token is consumed either the first time coming from the source or repeatedly coming from the token buffer. The macro is located before the return statement of a function that provides the parser with tokens. The current token is stored in the local variable yyToken:

```
TOKENOP
RETURN yyToken;
```

- The macro BEFORE_TRIAL can be used to execute statements before the start of every trial parsing.
- The macro AFTER_TRIAL can be used to execute statements after the end of every trial parsing. For example one application is to undo side-effects of trial parsing that fails.
- The macro ERROR can be used to execute statements in case of a syntax error before the execution of the builtin error recovery. Example:

```
  # define ERROR IO.WriteS (StdError, "syntax error at "); \
          Position.WritePosition (StdError, Attribute.Position);
```

- If the preprocessor symbol NO_RECOVER is defined then the builtin error recovery is excluded from the generated parser. The parser will terminate upon the first syntax error.

8.4.2. Scanner Interface

A generated parser needs the following objects from a module called Scanner:

```
DEFINITION MODULE <Scanner>;

IMPORT Position;

TYPE      tScanAttribute = RECORD Position: Position.tPosition; END;
VAR       Attribute      : tScanAttribute;
PROCEDURE ErrorAttribute (Token: CARDINAL; VAR Attribute: tScanAttribute);
PROCEDURE GetToken       (): INTEGER;

END <Scanner>.
```

- The procedure GetToken is repeatedly called by the parser in order to receive a stream of tokens. Every call has to return the internal integer representation of the "next" token. The end of the input stream (end of file) is indicated by a value of zero. The integer values returned by the procedure GetToken have to lie in a range between zero and the maximal value defined in the grammar. This condition is not checked by the parser and values outside of this range may lead to undefined behaviour.
- Additional properties or attributes of tokens are communicated from the scanner to the parser via the global variable Attribute. For tokens with additional properties like e. g. numbers or identifiers, the procedure GetToken has to supply the value of this variable as side-effect. The type of this variable can be chosen freely as long as it is an extension of the record type tScanAttribute.
- The variable Attribute must have a field called Position which describes the source coordinates of the current token. It has to be computed as side-effect of the procedure GetToken. In case of

syntax errors this field is passed as argument to the error handling routines.

- The type tParsAttribute must be a record type with at least one field called Scan of type tScanAttribute. Additional properties or attributes of tokens are transferred from the global variable Attribute to this field. See section 2.9 for an example.
- During automatic error repair a parser may insert tokens. In this case the parser calls the procedure ErrorAttribute to ask for the additional properties or attributes of an inserted token which is given by the argument Token. The procedure should return in the second argument called pAttribute default values for the additional properties of this token.

  Example:

```
PROCEDURE ErrorAttribute (Token: INTEGER; VAR pAttribute: tScanAttribute);
BEGIN
   pAttribute.Position := Attribute.Position;
   CASE Token OF
   | (* Ident   *) 1: pAttribute.Ident   := NoIdent;
   | (* Integer *) 2: pAttribute.Integer := 0      ;
   | (* Real    *) 3: pAttribute.Real    := 0.0    ;
   ELSE
   END;
END ErrorAttribute;
```

8.4.3. Error Interface

In case of syntax errors, the generated parser calls procedures in order to provide information about the position of the error, the set of expected tokens, and the behaviour of the repair and recovery mechanism. These procedures are conveniently implemented in a separate error handling module called Errors. The information provided by the parser may be stored or processed in an arbitrary way. A prototype error handling module is contained in the library *Reuse* (files reuse/src/Errors.md and reuse/src/Errors.mi) whose procedures immediately print the information passed as arguments.

```
DEFINITION MODULE Errors;

FROM SYSTEM     IMPORT ADDRESS;
FROM Position   IMPORT tPosition;

CONST
   SyntaxError          = 1     ;       (* error codes          *)
   ExpectedTokens       = 2     ;
   RestartPoint         = 3     ;
   TokenInserted        = 4     ;
   TokenFound           = 9     ;

   Error                = 3     ;       (* error classes        *)
   Repair               = 5     ;
   Information          = 7     ;

   String               = 7     ;       (* info classes         *)
   Array                = 8     ;

PROCEDURE ErrorMessage  (ErrorCode, ErrorClass: CARDINAL; Position: tPosition);
PROCEDURE ErrorMessageI (ErrorCode, ErrorClass: CARDINAL; Position: tPosition;
                         InfoClass: CARDINAL; Info: ADDRESS);
PROCEDURE Message       (ErrorText: ARRAY OF CHAR; ErrorClass: CARDINAL;
                         Position: tPosition);

END Errors.
```

- There are five messages a generated parser may report. They are encoded by the first group of the above constant definitions. The messages are classified according to the second group of constant definitions.
- The procedure ErrorMessage is used by the parser to report a message, its class, and its source position. It is used for syntax errors and recovery locations (restart points).
- The procedure ErrorMessageI is like the procedure ErrorMessage with additional Information. The latter is characterized by a class or type indication (InfoClass) and an untyped pointer (Info). Two types of additional information are used by the parser. During error repair, tokens might be inserted. These are reported one by one and they are classified as Array (ARRAY OF CHAR). At every syntax error the set of legal or expected tokens is reported using the classification String (tString).
- The procedure Message is similar to the procedure ErrorMessage. The first argument specifies a message text instead of an error code. This procedure might be called only if the reparsing feature is used.

8.4.4. Parser Driver

A main program is necessary for the test of a generated parser. The parser generator can provide a minimal main program in the file <Parser>Drv.mi which can serve as test driver. It has the following contents:

```
MODULE <Parser>Drv;

FROM Parser     IMPORT <Parser>, Close<Parser>;
FROM IO         IMPORT CloseIO;

BEGIN
   IF <Parser> () = 0 THEN END;
   Close<Parser>;
   CloseIO;
END <Parser>Drv.
```

8.5. Ada

8.5.1. Parser Interface

The parser interface consists of two parts: While the objects specified in the external interface can be used from outside the parser, the objects of the internal interface can be used only within a language description. The external parser interface in the file <Parser>.ads has the following contents:

```
package <Parser> is
                                        -- named constants for start symbols
yy<start_symbol_1>      : constant Integer := 1;
yy<start_symbol_2>      : constant Integer := 2;
...
                                        -- named constants for nonterminals
yyNT<nonterminal_1>     : constant Integer := 71;
yyNT<nonterminal_2>     : constant Integer := 72;
...
yyDebug                 : Boolean := False;

procedure Beginparser   ;
function  <Parser>      return Integer;
function  <Parser>2     (yyStartSymbol: Integer) return Integer;
procedure Closeparser   ;
function  TokenName     (Token: Integer) return String;

end <Parser>;
```

- The procedures <Parser> and <Parser>2 are the generated parsing routines. The argument StartSymbol of <Parser>2 specifies the start symbol to be used for parsing. Its value has to be one of the named constants yy<start_symbol_i>. The procedure <Parser> uses as start symbol the start symbol defined first in the grammar or the first nonterminal if no start symbols have been defined. The procedures return the number of syntax errors. A return value of zero indicates a syntactically correct input.
- For every nonterminal that is specified as start symbol a named constant is defined whose name consists of the name of the nonterminal with the prefix 'yy'. These constants are legal arguments for the procedure <Parser>2.
- For every nonterminal a named constant is defined whose name consists of the name of the nonterminal with the prefix 'yyNT'.
- The contents of the target code section named BEGIN is put into a procedure called Begin<Parser>. This procedure is called automatically upon every invocation of the procedure <Parser>. It can also be called explicitly by the user.

- The contents of the target code section named CLOSE is put into a procedure called Close<Parser>. It has to be called explicitly by the user when necessary.
- The variable yyDebug controls the trace of parsing actions. This trace is useful for debugging a parser. A trace is printed when the preprocessor directive

  ```
  # define YYDEBUG
  ```

  is included in the GLOBAL section and the variable yyDebug is set to true.
- The procedure TokenName provides a mapping from the internal representation of tokens to the external representation as given in the grammar specification. It maps integers to strings and it is used for example by the parser to provide expressive messages for the error handling module.

The internal parser interface consists of the following objects:

```
# define  yyInitStackSize  100
# define  yyInitBufferSize 100
# define  YYACCEPT
# define  YYABORT

          yyNonterminal : Integer;
          BufferPosition: Integer;

procedure SemActions    (Actions : Boolean);
procedure ErrorMessages (Messages: Boolean);
procedure MemoryClear   (Position: Integer);
function  GetLookahead  (k: Integer) return Integer;
procedure GetAttribute  (k: Integer; Attribute: out <Scanner>.tScanAttribute);
procedure BufferOn      (Actions, Messages: Boolean) return Integer;
procedure BufferOff     return Integer;
procedure ReParse       (StartSymbol: Integer; From, To: Integer;
                         Actions, Messages: Boolean) return Integer;
procedure BufferClear   ;
procedure yyOpenTrace   (FileName: String);

# define  TOKENOP
# define  BEFORE_TRIAL
# define  AFTER_TRIAL
# define  ERROR
# define  NO_RECOVER
```

Some of the objects are implemented differently using macros instead of functions.

- The initial size of the parser stack is defined by the value of the preprocessor symbol yyInitStackSize with a default of 100. The stack size is increased automatically when necessary. The initial stack size can be changed by including a preprocessor directive in the GLOBAL section such as:

  ```
  # define yyInitStackSize 200
  ```

- The initial size of the token buffer for trial parsing and reparsing is defined by the value of the preprocessor symbol yyInitBufferSize with a default of 100. The buffer size is increased automatically when necessary. The initial buffer size can be changed by including a preprocessor directive in the GLOBAL section such as:

  ```
  # define yyInitBufferSize 50
  ```

- The statement YYACCEPT can be used in semantic actions. It terminates the current invocation of the internal parsing procedure with the current count of errors.
- The statement YYABORT can be used in semantic actions. It terminates the current invocation of the internal parsing procedure with the current count of errors increased by one.
- The variable yyNonterminal holds the nonterminal of the left-hand side of the rule which has been reduced last. This variable can be accessed in semantic predicates or in semantic actions and this may be of good use in determining whether a trial parse needs to be done. The variable takes on values of enumerated constants generated for every nonterminal whose names are constructed by prefixing the name of the nonterminal with *yyNT*. Example:

```
yyNTdeclaration
yyNTstatement
yyNTexpression
```

- The procedure yyOpenTrace selects the file used for output of the trace information. The default value is Text_Io.Standard_Output. This value can be changed by calls of yyOpenTrace in the BEGIN section or in semantic actions.
- The procedure SemActions controls the execution of conditional semantic actions without a selection mask (see section 6.3.). The values of the argument Actions mean: true = enable, false = disable. The selection remains valid until a new call of the procedure or the termination of the current invocation of the internal parsing procedure.
- The procedure ErrorMessages controls the reporting of error messages by the parser. The values of the argument Messages mean: true = enable, false = disable. The selection remains valid until a new call of the procedure or the termination of the current invocation of the internal parsing procedure.
- The procedure MemoryClear is useful when the option -b is set to instruct the generated parser to memorize and reuse the results of previous trial parses. These results are stored at the positions in the token buffer where trial parses started. A call of MemoryClear clears a memorized result at the position in the token buffer given by the argument Position. The values for the position can be obtained from the variable BufferPosition (see below).
- The function GetLookahead returns the k-th lookahead token.
- The procedure GetAttribute returns via its second argument called Attribute the additional properties or attributes of the k-th lookahead token.
- The function BufferOn switches the parser to the mode buffer. This is only possible if the parser is in the mode standard, otherwise the mode remains unchanged. The arguments Actions and Messages control the execution of semantic actions and the reporting of error messages in the same way as the procedures SemActions and ErrorMessages. The function returns the position of the current lookahead token in the token buffer.
- The function BufferOff switches the parser to the mode standard. This is only possible if the parser is in the mode buffer, otherwise the mode remains unchanged. The selection for Actions and Messages is reset to the state before the call of BufferOn. The function returns the position of the current lookahead token in the token buffer.
- The variable BufferPosition holds the position of the current lookahead token in the token buffer. Its value can be read at any time while parsing in the mode buffer in order to request positions of arbitrary tokens in the token buffer.

- The function ReParse initiates parsing in the mode reparse using the start symbol StartSymbol. The contents of the token buffer from (including) buffer position From up to (excluding) buffer position To is parsed again according to the start symbol. The arguments Actions and Messages control the execution of semantic actions and the reporting of error messages in the same way as the procedures SemActions and ErrorMessages. The function returns the number of syntax errors found during its invocation.
- The procedure BufferClear signals that the contents of the token buffer can be deleted. The tokens will not be deleted immediately but when it is safe to do this. It is important to tell the parser when buffered tokens can be deleted, because otherwise all remaining input tokens will be buffered. Therefore every call of BufferOn should be accompanied by a call of BufferClear.
- The macro TOKENOP can be used to execute a piece of user code just before every token is consumed by the parser. This feature works only if the parser uses the token buffer. This is the case if trial parsing is activated or one of the following routines is used: GetLookahead, GetAttribute, BufferOn, BufferOff, ReParse, or BufferClear. Then the macro is executed every time a token is consumed either the first time coming from the source or repeatedly coming from the token buffer. The macro is located before the return statement of a function that provides the parser with tokens. The current token is stored in the local variable yyToken:

```
TOKENOP
return yyToken;
```

- The macro BEFORE_TRIAL can be used to execute statements before the start of every trial parsing.
- The macro AFTER_TRIAL can be used to execute statements after the end of every trial parsing. For example one application is to undo side-effects of trial parsing that fails.
- The macro ERROR can be used to execute statements in case of a syntax error before the execution of the builtin error recovery.
- If the preprocessor symbol NO_RECOVER is defined then the builtin error recovery is excluded from the generated parser. The parser will terminate upon the first syntax error.

8.5.2. Scanner Interface

A generated parser needs the following objects from a module called Scanner:

```
with Position;

package <Scanner> is

type      tScanAttribute is record Position: Position.tPosition; end record;
          Attribute      : tScanAttribute;
procedure ErrorAttribute (Token: Integer; Attribute: out tScanAttribute);
function  GetToken       return Integer;

end <Scanner>;
```

- The procedure GetToken is repeatedly called by the parser in order to receive a stream of tokens. Every call has to return the internal integer representation of the "next" token. The end of the input stream (end of file) is indicated by a value of zero. The integer values returned by the procedure GetToken have to lie in a range between zero and the maximal value defined in the grammar. This condition is not checked by the parser and values outside of this range may lead to undefined behaviour.

- Additional properties or attributes of tokens are communicated from the scanner to the parser via the global variable Attribute. For tokens with additional properties like e. g. numbers or identifiers, the procedure GetToken has to supply the value of this variable as side-effect. The type of this variable can be chosen freely as long as it is an extension of the record type tScanAttribute.
- The variable Attribute must have a field called Position which describes the source coordinates of the current token. It has to be computed as side-effect of the procedure GetToken. In case of syntax errors this field is passed as argument to the error handling routines.
- The type tParsAttribute must be a record type with at least one field called Scan of type tScanAttribute. Additional properties or attributes of tokens are transferred from the global variable Attribute to this field. See section 2.9 for an example.
- During automatic error repair a parser may insert tokens. In this case the parser calls the procedure ErrorAttribute to ask for the additional properties or attributes of an inserted token which is given by the argument Token. The procedure should return in the second argument called pAttribute default values for the additional properties of this token.

  Example:

```
procedure ErrorAttribute (Token: Integer; pAttribute: out tScanAttribute) is
begin
   pAttribute.Position := Attribute.Position;
   case Token is
   when 1 => -- Ident
      pAttribute.Ident.Ident     := NoIdent;
   when 2 => -- Integer
      pAttribute.IntConst.Value  := 0      ;
   when 3 => -- Real
      pAttribute.RealConst.Value := 0.0    ;
   when others => null;
   end case;
end ErrorAttribute;
```

8.5.3. Error Interface

In case of syntax errors, the generated parser calls procedures in order to provide information about the position of the error, the set of expected tokens, and the behaviour of the repair and recovery mechanism. These procedures are conveniently implemented in a separate error handling module called Errors. The information provided by the parser may be stored or processed in an arbitrary way. A prototype error handling module is contained in the library *Reuse* (files reuse/ada/errors.ads and reuse/ada/errors.adb) whose procedures immediately print the information passed as arguments.

```
with Position; use Position;

package Errors is

SyntaxError     : constant Integer := 1 ;       -- error codes
ExpectedTokens  : constant Integer := 2 ;
RestartPoint    : constant Integer := 3 ;
TokenInserted   : constant Integer := 4 ;
TokenFound      : constant Integer := 9 ;

Error           : constant Integer := 3 ;       -- error classes
Repair          : constant Integer := 5 ;
Information     : constant Integer := 7 ;

cString         : constant Integer := 7 ;       -- info classes

procedure ErrorMessage  (ErrorCode, ErrorClass: Integer; Position: tPosition);
procedure ErrorMessageI (ErrorCode, ErrorClass: Integer; Position: tPosition;
                         InfoClass: Integer; Info: Address);
procedure Message (ErrorText: String; ErrorClass: Integer; Position: tPosition);

end Errors;
```

- There are five messages a generated parser may report. They are encoded by the first group of the above constant definitions. The messages are classified according to the second group of constant definitions.
- The procedure ErrorMessage is used by the parser to report a message, its class, and its source position. It is used for syntax errors and recovery locations (restart points).
- The procedure ErrorMessageI is like the procedure ErrorMessage with additional Information. The latter is characterized by a class or type indication (InfoClass) and an untyped pointer (Info). During error repair, tokens might be inserted. These are reported one by one and they are classified as cString (String). At every syntax error the erroneous token and the set of legal or expected tokens is reported using the classification cString, too.
- The procedure Message is similar to the procedure ErrorMessage. The first argument specifies a message text instead of an error code. This procedure might be called only if the reparsing feature is used.

8.5.4. Parser Driver

A main program is necessary for the test of a generated parser. The parser generator can provide a minimal main program in the file <Parser>drv.adb which can serve as test driver. It has the following contents:

```
with <Parser>; use <Parser>;

procedure <Parser>Drv is

ErrorCount : Integer;

begin
   ErrorCount := <Parser>.parser;
   Close<Parser>;
end <Parser>Drv;
```

8.6. Eiffel

8.6.1. Parser Interface

The parser interface consists of two parts: While the objects specified in the external interface can be used from outside the parser, the objects of the internal interface can be used only within a language description. The file <Parser>.e contains the class <Parser> which offers the following features of the external interface:

```
class <Parser>

creation make

feature

yy<start_symbol_1>: INTEGER is 1
yy<start_symbol_1>: INTEGER is 2
...

make            is
SetDebug        (value: BOOLEAN) is
TokenName       (index: INTEGER): STRING is
BeginParser     is
Parser          : INTEGER is
Parser2         (StartSymbol: INTEGER): INTEGER is
CloseParser     is
```

- The procedure make instantiates a parser object and performs the necessary initializations. For example, the tables are read in from a file named <Parser>.txt.
- The procedures Parser and Parser2 are the generated parsing routines. The argument StartSymbol of Parser2 specifies the start symbol to be used for parsing. Its value has to be one of the named constants yy<start_symbol_i>. The procedure Parser uses as start symbol the start symbol defined first in the grammar or the first nonterminal if no start symbols have been defined. The procedures return the number of syntax errors. A return value of zero indicates a syntactically correct input.
- For every nonterminal that is specified as start symbol a named constant is defined whose name consists of the name of the nonterminal with the prefix 'yy'. These constants are legal arguments for the procedure Parser2.
- The contents of the target code section named BEGIN is put into a procedure called BeginParser. This procedure has to be called by the user when necessary.
- The contents of the target code section named CLOSE is put into a procedure called Close@Parser. It has to be called explicitly by the user when necessary.
- The procedure SetDebug controls the trace of parsing actions. This trace is useful for debugging a parser. A trace is printed only if the parser class is compiled with the switch debug enabled and after the procedure SetDebug has been called with the argument TRUE.
- The procedure TokenName provides a mapping from the internal representation of tokens to the external representation as given in the grammar specification. It maps integers to strings and it is used for example by the parser to provide expressive messages for the error handling module.

The internal parser interface consists of the following objects:

```
# define  YYACCEPT
# define  YYABORT

yyNonterminal   : INTEGER
BufferPosition  : INTEGER
yyTrace         : rFILE

SemActions      (Actions : BOOLEAN)
ErrorMessages   (Messages: BOOLEAN)
MemoryClear     (Position: INTEGER)
GetLookahead    (k: INTEGER): INTEGER
GetAttribute    (k: INTEGER): ScanAttribute
BufferOn        (Actions, Messages: BOOLEAN): INTEGER
BufferOff       : INTEGER
ReParse         (StartSymbol: INTEGER; From, To: INTEGER;
                 Actions, Messages: BOOLEAN): INTEGER
BufferClear

# define  TOKENOP
# define  BEFORE_TRIAL
# define  AFTER_TRIAL
# define  ERROR
# define  NO_RECOVER
```

Some of the objects are implemented differently using macros instead of functions.

- The statement YYACCEPT can be used in semantic actions. It terminates the current invocation of the internal parsing procedure with the current count of errors.
- The statement YYABORT can be used in semantic actions. It terminates the current invocation of the internal parsing procedure with the current count of errors increased by one.
- The variable yyNonterminal holds the nonterminal of the left-hand side of the rule which has been reduced last. This variable can be accessed in semantic predicates or in semantic actions and this may be of good use in determining whether a trial parse needs to be done. The variable takes on values of enumerated constants generated for every nonterminal whose names are constructed by prefixing the name of the nonterminal with *yyNT*. Example:

  ```
  yyNTdeclaration
  yyNTstatement
  yyNTexpression
  ```

- The variable yyTrace selects the file used for output of the trace information. The default value is stdout. This value can be changed by assignments in the BEGIN section or in semantic actions.
- The procedure SemActions controls the execution of conditional semantic actions without a selection mask (see section 6.3.). The values of the argument Actions mean: true = enable, false = disable. The selection remains valid until a new call of the procedure or the termination of the current invocation of the internal parsing procedure.
- The procedure ErrorMessages controls the reporting of error messages by the parser. The values of the argument Messages mean: true = enable, false = disable. The selection remains valid until a new call of the procedure or the termination of the current invocation of the internal parsing procedure.
- The procedure MemoryClear is useful when the option -b is set to instruct the generated parser to memorize and reuse the results of previous trial parses. These results are stored at the positions in the token buffer where trial parses started. A call of MemoryClear clears a memorized result at

the position in the token buffer given by the argument Position. The values for the position can be obtained from the variable BufferPosition (see below).

- The function GetLookahead returns the k-th lookahead token.
- The procedure GetAttribute returns via its second argument called Attribute the additional properties or attributes of the k-th lookahead token.
- The function BufferOn switches the parser to the mode buffer. This is only possible if the parser is in the mode standard, otherwise the mode remains unchanged. The arguments Actions and Messages control the execution of semantic actions and the reporting of error messages in the same way as the procedures SemActions and ErrorMessages. The function returns the position of the current lookahead token in the token buffer.
- The function BufferOff switches the parser to the mode standard. This is only possible if the parser is in the mode buffer, otherwise the mode remains unchanged. The selection for Actions and Messages is reset to the state before the call of BufferOn. The function returns the position of the current lookahead token in the token buffer.
- The variable BufferPosition holds the position of the current lookahead token in the token buffer. Its value can be read at any time while parsing in the mode buffer in order to request positions of arbitrary tokens in the token buffer.
- The function ReParse initiates parsing in the mode reparse using the start symbol StartSymbol. The contents of the token buffer from (including) buffer position From up to (excluding) buffer position To is parsed again according to the start symbol. The arguments Actions and Messages control the execution of semantic actions and the reporting of error messages in the same way as the procedures SemActions and ErrorMessages. The function returns the number of syntax errors found during its invocation.
- The procedure BufferClear signals that the contents of the token buffer can be deleted. The tokens will not be deleted immediately but when it is safe to do this. It is important to tell the parser when buffered tokens can be deleted, because otherwise all remaining input tokens will be buffered. Therefore every call of BufferOn should be accompanied by a call of BufferClear.
- The macro TOKENOP can be used to execute a piece of user code just before every token is consumed by the parser. This feature works only if the parser uses the token buffer. This is the case if trial parsing is activated or one of the following routines is used: GetLookahead, GetAttribute, BufferOn, BufferOff, ReParse, or BufferClear. Then the macro is executed every time a token is consumed either the first time coming from the source or repeatedly coming from the token buffer. The macro is located at the end of a function that provides the parser with tokens.

```
TOKENOP
```

- The macro BEFORE_TRIAL can be used to execute statements before the start of every trial parsing.
- The macro AFTER_TRIAL can be used to execute statements after the end of every trial parsing. For example one application is to undo side-effects of trial parsing that fails.
- The macro ERROR can be used to execute statements in case of a syntax error before the execution of the builtin error recovery.
- If the preprocessor symbol NO_RECOVER is defined then the builtin error recovery is excluded from the generated parser. The parser will terminate upon the first syntax error.

- Subclasses of the predefined support class Attribute should be specified in order to define the attributes needed for nonterminal symbols. See section 2.9 for an example.

8.6.2. Scanner Interface

A generated parser needs the following features from a class called <Scanner>:

```
class <Scanner>

EofToken        : INTEGER is 0
Attribute       : ScanAttribute

BeginScanner    is
GetToken        : INTEGER is
GetWord         : STRING is
ErrorAttribute  (Token: INTEGER): ScanAttribute is
SetAttribute    (value: ScanAttribute) is
```

- The procedure make instantiates a scanner object and performs the necessary initializations.
- The procedure GetToken is repeatedly called by the parser in order to receive a stream of tokens. Every call has to return the internal integer representation of the "next" token. The end of the input stream (end of file) is indicated by a value of zero. The integer values returned by the procedure GetToken have to lie in a range between zero and the maximal value defined in the grammar. This condition is not checked by the parser and values outside of this range may lead to undefined behaviour.
- Additional properties or attributes of tokens are communicated from the scanner to the parser via the feature Attribute. For tokens with additional properties like e. g. numbers or identifiers, the procedure GetToken has to supply the value of this feature as side-effect. The class of this feature can be chosen freely as long as it is a subclass of the class ScanAttribute.
- The procedure SetAttribute is needed by the parser to store values in the feature Attribute. This is necessary during backtracking in order to correctly backup to previous tokens.
- During automatic error repair a parser may insert tokens. In this case the parser calls the procedure ErrorAttribute to ask for the additional properties or attributes of an inserted token which is given by the argument Token. The procedure should return default values for the additional properties of this token.

  Example:

```
ErrorAttribute (Token: INTEGER): ScanAttribute is
   local
      IdentAttr         : IdentAttribute;
      IntegerAttr       : IntegerAttribute;
      RealAttr          : RealAttribute;
   do
      inspect Token
         when 1: !! IdentAttr.make (NoIdent); Result := IdentAttr  ;
         when 2: !! IntegerAttr.make (0)    ; Result := IntegerAttr;
         when 3: !! RealAttr.make (0.0)     ; Result := RealAttr   ;
      else       !! Result.make;
      end;
      Result.SetPosition (yyScanAttribute.Line, yyScanAttribute.Column);
   end;
```

8.6.3. Error Interface

In case of syntax errors, the generated parser calls procedures in order to provide information about the position of the error, the set of expected tokens, and the behaviour of the repair and recovery mechanism. These procedures are conveniently implemented in a separate error handling class called Errors. The information provided by the parser may be stored or processed in an arbitrary way. The parser generator can provide a prototype error handling class in the file errors.e whose procedures immediately print the information passed as arguments. This error handling class is provided when option -f is set (see also section: 7.4.5).

- There are seven messages a generated parser may report. They are encoded by the first group of the constant definitions. The messages are classified according to the second group of constant definitions.
- The procedure make initializes an Errors object.
- The procedure ErrorMessage is used by the parser to report a message, its class, and its source position. It is used for syntax errors and recovery locations (restart points).
- The procedure ErrorMessageI is like the procedure ErrorMessage with additional Information. The latter is characterized by a class indication (InfoClass) and an arbitrary object (Info). During error repair tokens might be inserted. These are reported one by one and they are classified as IsString. At every syntax error the set of legal or expected tokens is reported using the classification IsString, too.
- The procedure Message is similar to the procedure ErrorMessage. The first argument specifies a message text instead of an error code. This procedure might be called only if the reparsing

```
class Errors

creation make

feature

SyntaxError     : INTEGER is unique     -- error codes
ExpectedTokens  : INTEGER is unique
RestartPoint    : INTEGER is unique
TokenInserted   : INTEGER is unique
WrongParseTable : INTEGER is unique
TokenFound      : INTEGER is unique
FoundExpected   : INTEGER is unique

Fatal           : INTEGER is unique     -- error classes
Error           : INTEGER is unique
Repair          : INTEGER is unique
Information     : INTEGER is unique

IsString        : INTEGER is unique     -- info classes

BRIEF           : BOOLEAN
FIRST           : BOOLEAN
TRUNCATE        : BOOLEAN

make           is
ErrorMessage    (ErrorCode, ErrorClass, Line, Column: INTEGER) is
ErrorMessageI   (ErrorCode, ErrorClass, Line, Column, InfoClass: INTEGER;
                 Info: ANY) is
Message         (ErrorText: STRING; ErrorClass: INTEGER; Position: Position) is
```

feature is used.

8.6.4. Parser Driver

A main program is necessary for the test of a generated parser. The parser generator can provide a minimal main class in the file <Parser>drv.e which can serve as test driver. It has the following contents:

```
class <Parser>Drv

creation main

feature

main is
   local
      Parser    : Parser
      ErrorCount: INTEGER
   do
      !! Parser.make
      Parser.BeginParser
      ErrorCount := Parser.Parser
      Parser.CloseParser
   end

end
```

8.6.5. Support Classes

A parser generated in the language Eiffel needs several support classes which are provided the directory reuse/eiffel. These classes are:

Attribute
ScanAttribute
Errors
Position
rFile
rSystem

The classes can be found in files whose names consist of the class name and the suffix *.e*. The class Attribute serves as a super class for classes that define the structure of the attributes for the S-attribution mechanism of *Lark*. The class ScanAttribute is a subclass of the class Attribute and it serves as a super class for classes that define the attributes of tokens or terminals whose values are provided by the scanner. Appropriate subclasses of these two classes should be defined according to the user's need. The class Errors contains the procedures for error handling described in the previous section. It can also be adapted to the user's need. The class Position represents the source positions of tokens. The classes rFile and rSystem are for the adaption to system specific properties.

9. Error Recovery

The parser generator always includes data and algorithms for the handling of syntax errors in a generated parser. The error handling includes error recovery, error reporting, and error repair. It is generated fully automatic - there are no instructions necessary to achieve this behaviour. The error messages use the terminal symbols of the grammar. Therefore the use of self explanatory identifiers or strings for the denotation of terminals is recommended.

Lark uses the complete backtrack-free method described in [Röh76, Röh80, Röh82]. Every incorrect input is "virtually" transformed into a syntactically correct program with the consequence of executing a "correct" sequence of semantic actions, only. This has the advantage that the following compiler phases like semantic analysis do not have to deal with syntax errors. The library *Reuse* [Grod, Groe] provides a prototype error module called *Errors* which just prints messages as shown below. If an other format or a different behaviour of the error handling module is desired, then it can be adapted easily to the ideas of the user. The body of the module contains three (C preprocessor) variables that control the style of the error messages:

BRIEF	summarize syntax errors in one error message instead of several messages
FIRST	report only the first error message on a line instead of all messages
TRUNCATE	truncate additional information for messages (such as the set of expected symbols) to around 25 characters

Example: The following Pascal program contains two syntax errors:

```
program test (output);
begin
   if (a = b] write (a;
end.
```

If all three preprocessor variables are undefined then the following messages are reported:

```
3, 13: Error       syntax error
3, 13: Information token found    : ]
3, 13: Information expected tokens: ) = + - <> <= >= < > IN OR * / DIV MOD AND
3, 15: Information restart point
3, 15: Repair      token inserted : )
3, 15: Repair      token inserted : THEN
3, 23: Error       syntax error
3, 23: Information token found    : ;
3, 23: Information expected tokens: , ) = + - : <> <= >= < > IN OR * / DIV MOD AND
3, 23: Repair      token inserted : )
```

If BRIEF is defined then this is compressed into two lines:

```
3, 13: Error       found/expected : ]/) = + - <> <= >= < > IN OR * / DIV MOD AND
3, 23: Error       found/expected : ;/, ) = + - : <> <= >= < > IN OR * / DIV MOD AND
```

If BRIEF and FIRST are defined then this results in just one line:

```
3, 13: Error       found/expected : ]/) = + - <> <= >= < > IN OR * / DIV MOD AND
```

If BRIEF, FIRST, and TRUNCATE are defined then this one line becomes even shorter:

```
3, 13: Error       found/expected : ]/) = + - <> <= >= < > IN OR * / ...
```

In all of the abbreviated styles the information about restart points or inserted tokens is suppressed and the messages reporting the found token and the set of expected tokens are combined into one message.

A syntax error is handled by the parser as follows:

- A syntax error is detected when the table of the parser does not indicate an action for the current values of the parser state and the lookahead token.

- The fact of a syntax error and its location are reported.
- The erroneous lookahead token is reported.
- All tokens that would be a legal continuation of the program at the current location are computed and reported.
- All tokens that can be used to safely continue with parsing are computed. A minimal sequence of tokens is skipped until one of these tokens is found. This token represents the recovery location.
- The recovery location or restart point is reported.
- Parsing continues in the so-called repair mode. The parser behaves as usual in this mode except that tokens are not requested from the scanner. Instead, a minimal sequence of tokens is synthesized to repair the error. The parser stays in this mode until the token at the recovery location can be accepted. The synthesized tokens are reported. The program can be regarded as repaired, if the skipped tokens are replaced by the synthesized ones. Upon leaving repair mode, parsing continues as usual.

10. Support for Debugging

Debugging of generated parsers is supported by a readable trace of the parsing actions and a graphic visualization.

10.1. Trace of Parsing Actions

A readable trace of the actions executed by the parser can be requested as an aid for debugging a generated parser. The trace is printed to a file selected by the variable yyTrace which defaults to standard output. It has to be enabled by a compile-time switch as well as by a run-time switch. The details are language dependent:

C	The parser must be compiled with the option -DYYDEBUG and the external variable <Parser>_Debug must be set to true (= 1).
C++	The parser must be compiled with the option -DYYDEBUG and the member yyDebug must be set to true (= 1).
Java	The preprocessor directive `# define YYDEBUG` must be included in the GLOBAL section and the variable yyDebug, which is exported by the class <Parser>, must be set to true.
Modula-2	The preprocessor directive `# define YYDEBUG` must be included in the GLOBAL section and the variable yyDebug, which is exported by the module <Parser>, must be set to TRUE.
Ada	The preprocessor directive `# define YYDEBUG` must be included in the GLOBAL section and the variable yyDebug, which is exported by the package <Parser>, must be set to True.
Eiffel	The class parser must be compiled with the switch *debug* enabled and the procedure SetDebug must be called with the argument TRUE.

For the explanation of the trace we use the grammar from section 5.3. Suppose the input of the generated parser is:

```
{
   T (i);
   T (i) ++;
   f (x);
}
```

Then the following trace is printed:

```
  # |Position|State|Mod|Lev|Action |Terminal and Lookahead or Rule

   1:   1,  1:    1  S      parse   for predicate in line 23, lookahead: {
   2:   2,  4:    1  S      shift   {, lookahead: IDENTIFIER
   3:   2,  4:    3  S      reduce  declaration_list :
   4:   2,  4:    6  S      dynamic decision 1
   5:   2,  4:    2  T   1    parse   for predicate in line 31, lookahead: IDENTIFIER
   6:   2,  4:    2  T   1    reduce  1_declaration_Trial_1 :
   7:   2,  6:    5  T   1    shift   IDENTIFIER, lookahead: (
   8:   2,  6:    7  T   1    dynamic decision 2
   9:   2,  6:    7  T   1    check   predicate in line 50, result = 1
  10:   2,  6:    7  T   1    reduce  TYPEDEFname : IDENTIFIER
  11:   2,  6:    5  T   1    reduce  declaration_specifier : TYPEDEFname
trial action
  12:   2,  7:    8  T   1    shift   (, lookahead: IDENTIFIER
  13:   2,  8:   10  T   1    shift   IDENTIFIER, lookahead: )
  14:   2,  8:   10  T   1    reduce  declarator : IDENTIFIER
trial action
  15:   2,  9:   14  T   1    shift   ), lookahead: ;
  16:   2,  9:   14  T   1    reduce  declarator : '(' declarator ')'
  17:   3,  4:   11  T   1    shift   ;, lookahead: IDENTIFIER
  18:   3,  4:   11  T   1    reduce  declaration : 1_declaration_Trial_1 declaration_specifier declarator
  19:   3,  4:    2  T   1    reduce  0_intern : declaration
  20:   3,  4:    2  T   1    accept  parse started at 5
  21:   2,  4:    6  S      check   predicate in line 31, result = 1
  22:   2,  4:    6  S      reduce  1_declaration_Trial_1 :
  23:   2,  6:    5  S      shift   IDENTIFIER, lookahead: (
  24:   2,  6:    7  S      dynamic decision 2
  25:   2,  6:    7  S      check   predicate in line 50, result = 1
  26:   2,  6:    7  S      reduce  TYPEDEFname : IDENTIFIER
  27:   2,  6:    5  S      reduce  declaration_specifier : TYPEDEFname
trial action
final action
  28:   2,  7:    8  S      shift   (, lookahead: IDENTIFIER
  29:   2,  8:   10  S      shift   IDENTIFIER, lookahead: )
  30:   2,  8:   10  S      reduce  declarator : IDENTIFIER
trial action
final action
  31:   2,  9:   14  S      shift   ), lookahead: ;
  32:   2,  9:   14  S      reduce  declarator : '(' declarator ')'
  33:   3,  4:   11  S      shift   ;, lookahead: IDENTIFIER
  34:   3,  4:   11  S      reduce  declaration : 1_declaration_Trial_1 declaration_specifier declarator '
  35:   3,  4:    6  S      reduce  declaration_list : declaration_list declaration
  36:   3,  4:    6  S      dynamic decision 1
  37:   3,  4:    2  T   1    parse   for predicate in line 31, lookahead: IDENTIFIER
  38:   3,  4:    2  T   1    reduce  1_declaration_Trial_1 :
  39:   3,  6:    5  T   1    shift   IDENTIFIER, lookahead: (
  40:   3,  6:    7  T   1    dynamic decision 2
  41:   3,  6:    7  T   1    check   predicate in line 50, result = 1
  42:   3,  6:    7  T   1    reduce  TYPEDEFname : IDENTIFIER
  43:   3,  6:    5  T   1    reduce  declaration_specifier : TYPEDEFname
trial action
  44:   3,  7:    8  T   1    shift   (, lookahead: IDENTIFIER
  45:   3,  8:   10  T   1    shift   IDENTIFIER, lookahead: )
  46:   3,  8:   10  T   1    reduce  declarator : IDENTIFIER
trial action
  47:   3, 10:   14  T   1    shift   ), lookahead: ++
  48:   3, 10:   14  T   1    reduce  declarator : '(' declarator ')'
  49:   3, 10:   11  T   1    fail    parse started at 37
  50:   3,  4:    6  S      check   predicate in line 31, result = 0
  51:   3,  4:    6  S      reduce  statement_list :
  52:   3,  6:    9  S      shift   IDENTIFIER, lookahead: (
  53:   3,  6:   18  S      dynamic decision 3
```

```
54:   3,  6:   18  S      check   predicate in line 50, result = 1
55:   3,  6:   18  S      reduce  TYPEDEFname : IDENTIFIER
56:   3,  7:   13  S      shift   (, lookahead: IDENTIFIER
57:   3,  8:   16  S      shift   IDENTIFIER, lookahead: )
58:   3,  8:   19  S      reduce  expression : IDENTIFIER
59:   3, 10:   17  S      shift   ), lookahead: ++
60:   3, 10:   17  S      reduce  expression : TYPEDEFname '(' expression ')'
61:   3, 12:   12  S      shift   ++, lookahead: ;
62:   3, 12:   12  S      reduce  expression : expression '++'
63:   4,  4:   12  S      shift   ;, lookahead: IDENTIFIER
64:   4,  4:   12  S      reduce  statement : expression ';'
65:   4,  4:    9  S      reduce  statement_list : statement_list statement
66:   4,  6:    9  S      shift   IDENTIFIER, lookahead: (
67:   4,  6:   18  S      dynamic decision 3
68:   4,  6:   18  S      check   predicate in line 50, result = 0
69:   4,  6:   18  S      check   predicate in line 51, result = 0
70:   4,  7:   18  S      shift   (, lookahead: IDENTIFIER
71:   4,  8:   20  S      shift   IDENTIFIER, lookahead: )
72:   4,  8:   19  S      reduce  expression : IDENTIFIER
73:   4,  9:   22  S      shift   ), lookahead: ;
74:   4,  9:   22  S      reduce  expression : IDENTIFIER '(' expression ')'
75:   5,  1:   12  S      shift   ;, lookahead: }
76:   5,  1:   12  S      reduce  statement : expression ';'
77:   5,  1:    9  S      reduce  statement_list : statement_list statement
78:   6,  1:    9  S      shift   }, lookahead: _EOF_
79:   6,  1:    9  S      reduce  compound_statement : '{' declaration_list statement_list '}'
80:   6,  1:    4  S      reduce  0_intern : compound_statement _EOF_
81:   6,  1:    4  S      accept  parse started at 1
```

The trace starts with a heading that explains the meaning of the different columns: The first column contains consecutive numbers for the parser actions. The column Position gives the source position of the current lookahead token which consists of a line and a column by default. The format depends on the procedure WritePosition of the module Position[s] which can be redefined. The column State contains the (external) number of the current parser state. The column Mod gives the current mode of the parser abbreviated by a letter:

S standard
T trial
B buffer
R reparse

The column Lev contains the recursion level of the internal parsing procedure. A blank entry indicates the first activation and a number n specifies that n+1 invocations are active. The column Action specifies one of seven possible actions which are indented according to the recursion level: shift, reduce, parse, accept, fail, check, or dynamic. The information following the action in the last column depends on the kind of the action:

shift — The parser always works with a lookahead of one token. A shift action consumes one token by pushing the current lookahead token on the parser stack. A new lookahead token is read by either calling the scanner procedure GetToken or by inspecting a token buffer during trial parsing and reparsing. The column Position gives the source position of the new lookahead token and the last column contains the pushed token as well as the new lookahead token.

reduce — The parser reduces or recognizes a certain grammar rule which follows in the last column.

parse — Parsing in one of the modes standard, trial, or reparse is initiated. The column State gives the (external) number of the start state used for this parse. In case of a trial parse the line number of the predicate that caused the trial parse is provided in the last

column. In case of a standard parse or reparsing the line where the start symbol is defined is regarded as a special kind of predicate and reported in the last column. Additionally, the current lookahead token is reported.

accept Parsing in one of the modes standard, trial, or reparse terminates successfully without any errors. The last column contains a reference to the parse action that initiated the parse.

fail Parsing in one of the modes standard, trial, or reparse terminates unsuccessfully (with errors). The last column contains a reference to the parse action that initiated the parse.

dynamic A dynamic decision which depends on the result of a predicate is necessary in order to find out how to continue parsing. The decision may lead to a reduce action for a certain grammar rule or to a shift action. The number given in the last column refers to the different decision schemes generated by *Lark*.

check A predicate is checked. The last column contains the line number of the predicate in the grammar and the result: 0 or F stand for false and 1 or T stand for true. A check action is completely covered by one trace line in case of semantic predicates or syntactic predicates with a terminal. However, syntactic predicates with a nonterminal trigger trial parsing and then a check action is preceded by a sequence of actions that trace the trial parsing. This sequence is enclosed either in a parse-accept pair or a parse-fail pair of actions.

Trial parsing as well as reparsing may be nested to arbitrary depth. This means that corresponding parse-accept pairs and parse-fail pairs may appear nested, too.

If option -b is set then the generated parser memorizes the result of previous trial parses and tries to avoid the unnecessary repetition of same trial parses. If a dynamic action with a nonterminal predicate is immediately followed by a check action then the parser used a memorized result of a previous trial parse and no parsing actions have been performed to obtain this result.

The state of the parser and the lookahead token determine the next action of the parser. In the example, the first action (line 2) is a shift of { with a transition to state 3. The transition to state 3 can be concluded from the column State in line 3. A reduce action can only take place when all symbols of the right-hand side of the rule to be reduced are on top of the parser stack. Symbols are pushed on the stack in two cases: A shift action pushes a terminal on the stack and a reduce action first pops as many symbols from the stack as there are symbols on the right-hand side of the rule and then it pushes the nonterminal on the left-hand side of the rule. For example at line 3 one token has been traced as shifted: { . The lookahead token is IDENTIFIER. This situation characterizes the beginning of a declaration_list. Accordingly, a reduce action is performed at line 3 which recognizes an empty sequence of tokens as a declaration_list by reducing the rule:

```
declaration_list : .
```

This reduce action pops zero symbols from the stack because there are zero symbols on the right-hand side of the rule. Then it pushes the nonterminal declaration_list on the stack. Now we have the symbols { and declaration_list on the stack and the token IDENTIFIER is still the current lookahead token.

10.2. Graphic Visualization

With option -5 a parser with graphic visualization is generated. This parser opens a window which displays the parser stack and offers a set of menue buttons for mouse activation. There are among

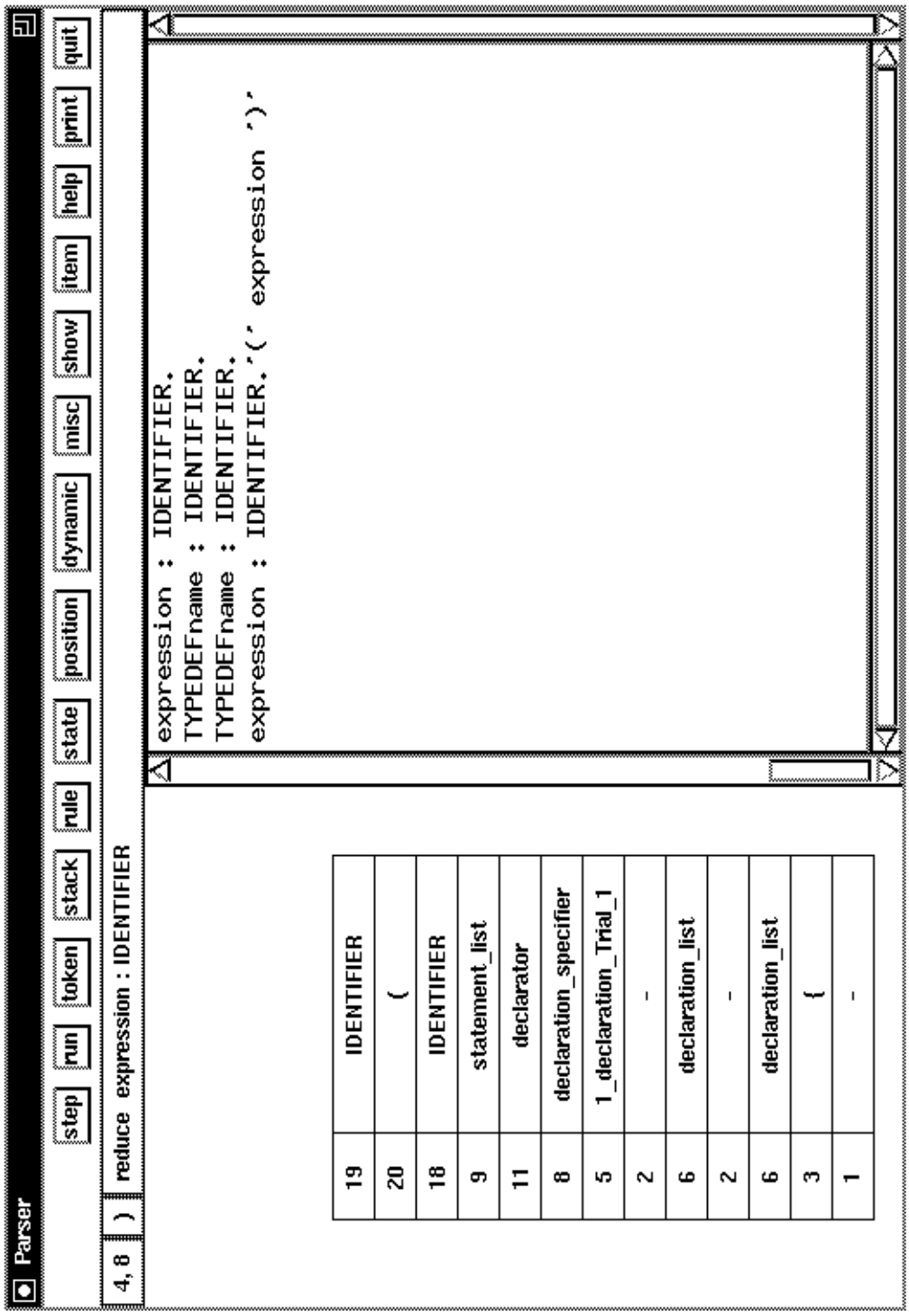
others commands for single stepping and the definition of various breakpoints. The help command explains all available commands. This parser variant can be regarded as a debugging tool with graphic user interface. With respect to semantic actions, trace, or linking with other modules this parser behaves as any other parser. Currently, parsers with visualization can be generated for the language C, only. For linking a parser with visualization the public domain package Tcl/Tk 8.4 is required. Under Unix linking is done with commands like:

```
cc -L/usr/X11/lib -ltcl8.4 -ltk8.4 -lX11 -lm .../libreuse.a
```

See the screenshot on the next page for an example of the visualizer window. The left window displays the stack and the right window currently displays the item set of the state on top of the stack. Every stack entry consists of a state number and the grammar symbol that caused the transition to this state.

The following is the online help information describing the meaning of the window and the menue buttons:

```
The top line contains:

the source position of the current lookahead token,
the current lookahead token,
the next action to be executed.

The left window displays the stack.
The right window displays various information.

The possible actions are:

shift      : the current lookahead token is pushed on the stack,
             the next token is read and becomes the new lookahead token
reduce     : a grammar rule is recognized,
             its right-hand side symbols are popped from the stack and
             its left-hand side nonterminal symbol is pushed on the stack
error      : a syntax error has been detected
accept     : parsing terminates (at EOF or by ACCEPT)
fail       : parsing terminates (by ABORT)
dynamic decision: a predicate is checked in order to determine the next action

The menu buttons:

step       : execute one parser action
run        : execute parser actions until a breakpoint is hit
break      : define a breakpoint (see below)
show       : display the defined breakpoints,
             delete a breakpoint by clicking on it with the left mouse button
item       : display the set of items for the state on top of the stack
help       : display this help information
print      : write the current picture of the stack in postscript format
             to the file named 'Parser.ps'
exit       : exit the program

Definition of breakpoints:

token      : stop when a certain token becomes the lookahead token
stack      : stop when a certain symbol is on top of the stack
rule       : stop when a certain grammar rule is reduced
```



Screenshot of the graphic visualization

```
state      : stop when a certain state is on top of the stack
position   : stop when a certain source position is reached
dynamic    : stop when a certain dynamic decision is evaluated
dynamic all: stop when any dynamic decision is evaluated
error      : stop when a syntax error has been detected
accept     : stop when parsing terminates without errors
fail       : stop when parsing terminates with errors

             For token, stack, and rule select an item in the right
             window by clicking on it with the left mouse button.
```

11. Usage and Options

NAME

lark − LALR(2) parser generator with backtracking

SYNOPSIS

lark [-options] [-l*directory*] [*file*]

DESCRIPTION

Lark is a parser generator for highly efficient parsers that generates parsers for LALR(2) and LR(1) grammars. Moreover, backtracking can be used to recognize even larger grammar classes such as non-LR(k) grammars. The grammar is read from the file given as argument or from standard input, if the file argument is missing. Each grammar rule may be associated with a semantic action consisting of arbitrary statements written in the target language. Whenever a grammar rule is recognized by the generated parser the associated semantic action is executed. A mechanism for S-attribution (synthesized attributes) is provided that allows for the communication between the semantic actions. Attributes can be accessed by symbolic names as well as by the $n notation.

In case of LR conflicts, derivation trees are printed that ease the location of the problem. Ambiguities in the grammar may be solved by specifying precedence and associativity for tokens and grammar rules or by the use of semantic and syntactic predicates. Syntactic errors are handled completely automatic by the generated parsers including error reporting, error recovery, and error repair. The generated parsers are table-driven.

The generated parser needs a scanner, an error handling module, and a few modules from a library of reusable modules. Errors detected during the analysis of the grammar are reported on standard error. Detailed explanations about LR conflicts can be requested with the options -v or -w. While option -v selects the explanation of all conflicts, option -w restricts the explanation to implicitly repaired conflicts and dynamically repaired conflicts (using predicates) and it suppresses the explanation of explicitly repaired conflicts (using precedence and associativity). The explanation is written to a file whose name is the name of the input file with the suffix ".dbg" or "Parser.dbg" in case of input comes from standard input.

OPTIONS

c generate C source code

\+ generate C++ source code

j generate Java source code

m generate Modula-2 source code (default)

3 generate Ada source code

e generate Eiffel 3 source code

a generate all = -dip (default)

d generate header file or definition module

i generate implementation part or module

p generate main program to be used as test driver

5 generate parser with graphical visualization

-f[*prefix*]
generate constant declarations for tokens in header file using prefix (default: t_)

g generate # line directives

: generate lines not longer than 80 characters

J report undefined tokens and multiply defined nonterminals as error (default: warning)

2 suppress reporting of multiply defined nonterminals

s suppress informations and warnings

r suppress elimination of LR(0) reductions

u use faster and larger terminal tables (default: slower and smaller)

o use faster and larger nonterminal tables (default: slower and smaller)

4 use tables to decrement stack pointers (default: inline code)

6 use alternate algorithm for table compression

b memorize previous trial parses and avoid repetition of same trial parses

v explain all LR conflicts in file with suffix .dbg

w explain implicitly and dynamically repaired conflicts, only

D explain new conflicts, only - old conflicts are read from file with suffix .cft, current set of conflicts is written to new version of this file, differences between previous and current runs are written to file with suffix .dlt

h print help information

n reduce the number of case labels in switch, case, or inspect statements by mapping so-called shift-reduce states to reduce states (increases run time a little bit but decreases code size, might be necessary due to compiler restrictions)

-n*number*
generate switch or case statements with at most number case labels (might be necessary due to compiler restrictions)

t print statistics about the generated parser

x print a list of terminals and their encoding

y print a readable version of the generated automaton (states and items)

z print a list of nonterminals and rules

0 construct an LALR(1) parser (based on an LR(0) automaton) (default)

1 construct an LR(1) parser (based on an LR(1) automaton)

01 construct an LALR(1) parser and extend it locally to LR(1) where necessary

-k2 construct an LALR(2) parser (which uses 2 tokens lookahead) (default: 1)

7 touch output files only if necessary

8 report storage consumption

-l*directory*
specify the directory where lark finds its data files

H print advanced help information

U explain explicitly repaired conflicts

V explain implicitly repaired conflicts

W explain LALR(2) repaired conflicts

X explain dynamically repaired conflicts

C disable generation of comments and rules

FILES

*.dbg	readable explanation of LR conflicts
*.cft	internal representation of LR conflicts
*.dlt	differences in grammar and conflicts wrt. previous run

if output is in C:

<Parser>.h	header file for the generated parser
<Parser>.c	body of the generated parser
<Parser>Drv.c	body of the parser driver (main program)
yySource	temporary file of visualizing parser
yy<Parser>.brk	temporary file of visualizing parser
yy<Parser>.itm	temporary file of visualizing parser
yy<Parser>.rul	temporary file of visualizing parser
yy<Parser>.sbl	temporary file of visualizing parser
Parser.tcl	procedure definitions for visualizing parser

if output is in C++:

<Parser>.h	header file for the generated parser
<Parser>.cxx	body of the generated parser
<Parser>Drv.cxx	body of the parser driver (main program)
yySource	temporary file of visualizing parser
yy<Parser>.brk	temporary file of visualizing parser
yy<Parser>.itm	temporary file of visualizing parser
yy<Parser>.rul	temporary file of visualizing parser
yy<Parser>.sbl	temporary file of visualizing parser
Parser.tcl	procedure definitions for visualizing parser

if output is in Java:

<Parser>.java	class of the generated parser

<Parser>Drv.java	parser driver (main program)

if output is in Modula-2:

<Parser>.md	definition module of the generated parser
<Parser>.mi	implementation module of the generated parser
<Parser>Drv.mi	implementation module of the parser driver

if output is in Ada:

<Parser>.ads	package (interface) of the generated parser
<Parser>.adb	package body of the generated parser
<Parser>drv.adb	package body of the parser driver

if output is in Eiffel:

<Parser>.e	class of the generated parser
<Parser>drv.e	class of the test driver (main program)
<Parser>.txt	tables controlling the parser (ASCII format)
errors.e	class of error handler
attribute.e	support class for the description of properties of nonterminals
scanattribute.e	support class for the description of properties of tokens
position.e	support class for the representation of source positions
rfile.e	support class extending the class FILE
rsystem.e	support class for system specific properties

Acknowledgement

I thank Dr. Thomas Herter for the stimulation of the implementation of trial parsing and for many fruitful discussions.

Appendix 1: Syntax of the Input Language

```
RULE

Grammar         : Decls RulePart
                .
Decls           :       /* empty */
                | Decls ScannerName
                | Decls ParserName
                | Decls Codes
                | Decls TokenPart
                | Decls PrecPart
                | Decls StartPart
                .
ScannerName     : 'SCANNER'
                | 'SCANNER' Identifier
                .
ParserName      : 'PARSER'
                | 'PARSER' Identifier
                .
Codes           : 'IMPORT' Action
                | 'EXPORT' Action
                | 'GLOBAL' Action
                | 'LOCAL'  Action
                | 'BEGIN'  Action
                | 'CLOSE'  Action
                .
TokenPart       : 'TOKEN' Tokens
                .
Tokens          :       /* empty */
                | Tokens Token Code Cost
                | Tokens Token Code Repr
                .
Token           : Identifier
                | String
                .
Code            :       /* empty */
                | Number
                | '=' Number
                .
Cost            :       /* empty */
                | ',' Number
                | ',' Number ',' String
                .
Repr            : ',' String
                | ',' String ',' Number
                .
PrecPart        : 'PREC' Precedences
                .
Precedences     :       /* empty */
                | Precedences 'LEFT'  Terminals
                | Precedences 'RIGHT' Terminals
                | Precedences 'NONE'  Terminals
                .
Terminals       :       /* empty */
                | Terminals Identifier
                | Terminals String
```

```
                .
StartPart       : 'START' Nonterminals
                .
Nonterminals    :       /* empty */
                | Nonterminals Identifier
                .
RulePart        : 'RULE' Productions
                .
Productions     : Identifier ':' Rules '.'
                | Productions Identifier ':' Rules '.'
                .
Rules           : Elements
                | Rules '|' Elements
                .
Elements        :       /* empty */
                | Elements Identifier
                | Elements Identifier '$' Identifier
                | Elements String
                | Elements String '$' Identifier
                | Elements Action
                | Elements Action '$' Identifier
                | Elements UCAction
                | Elements UCAction '$' Identifier
                | Elements 'PREC' Identifier
                | Elements 'PREC' String
                | Elements '?' Action
                | Elements '?' Identifier
                | Elements '?' String
                | Elements '?' '-' Action
                | Elements '?' '-' Identifier
                | Elements '?' '-' String
                .
                                        /* lexical grammar */
Identifier      : Letter
                | '_'
                | '\'
                | Identifier Letter
                | Identifier Digit
                | Identifier '_'
                .
Number          : Digit
                | Number Digit
                .
String          : "'" Characters "'"
                | '"' Characters '"'
                .
Action          : '{' TargetCode '}'    /*   conditional or final action */
                .
UCAction        : '[' TargetCode ']'    /* unconditional or trial action */
                .
TargetCode      :       /* empty */
                | TargetCode Character
                | TargetCode Attribute
                .
Attribute       : '$' Number
                | '$' '-' Number
                | '$$'
```

```
                | '$' Identifier
                .
Comment1        : '(*' Characters '*)'
                .
Comment2        : '/*' Characters '*/'
                .
Characters      :
                | Characters Character
                .
```

Appendix 2: Example: Desk Calculator in C (symbolic access)

```
GLOBAL {
typedef union { tScanAttribute Scan; int value; } tParsAttribute;
int regs [26], base;
}

TOKEN
        DIGIT   = 1
        LETTER  = 2
        '+'     = 43
        '-'     = 45
        '*'     = 42
        '/'     = 47
        '%'     = 37
        '\n'    = 10
        '='     = 61
        '('     = 40
        ')'     = 41

PREC
        LEFT    '+'     '-'
        LEFT    '*'     '/'     '%'
        LEFT    UNARY

RULE

list    :
        | list stat '\n'
        .
stat    : expr $e               { printf ("%d\n", $e.value);       }
        | LETTER $l '=' expr $e { regs [$l.Scan.value] = $e.value; }
        .
expr    : '(' expr $e ')'       { $$.value = $e.value;             }
        | expr $l '+' expr $r   { $$.value = $l.value + $r.value;  }
        | expr $l '-' expr $r   { $$.value = $l.value - $r.value;  }
        | expr $l '*' expr $r   { $$.value = $l.value * $r.value;  }
        | expr $l '/' expr $r   { $$.value = $l.value / $r.value;  }
        | expr $l '%' expr $r   { $$.value = $l.value % $r.value;  }
        | '-' expr $e           { $$.value = - $e.value;           } PREC UNARY
        | LETTER $l             { $$.value = regs [$l.Scan.value]; }
        | number $n             { $$.value = $n.value;             }
        .
number  : DIGIT $d              { $$.value = $d.Scan.value;
                                  base = $d.Scan.value == 0 ? 8 : 10; }
        | number $n DIGIT $d    { $$.value = base * $n.value + $d.Scan.value; }
        .
```

Appendix 3: Example: Desk Calculator in Modula-2 (numeric access)

```
GLOBAL {
FROM StdIO      IMPORT WriteI, WriteNl;
FROM Scanner    IMPORT tScanAttribute;
TYPE tParsAttribute = RECORD Scan: tScanAttribute; value: INTEGER; END;
VAR regs: ARRAY [0..25] OF INTEGER;
VAR base: INTEGER;
}

TOKEN
        DIGIT   = 1
        LETTER  = 2
        '+'     = 43
        '-'     = 45
        '*'     = 42
        '/'     = 47
        '%'     = 37
        '\n'    = 10
        '='     = 61
        '('     = 40
        ')'     = 41

PREC
        LEFT    '+'     '-'
        LEFT    '*'     '/'     '%'
        LEFT    UNARY

RULE

list    :
        | list stat '\n'
        .
stat    : expr            { WriteI ($1.value, 0); WriteNl;     }
        | LETTER '=' expr { regs [$1.Scan.value] := $3.value;  }
        .
expr    : '(' expr ')'    { $$.value := $2.value;              }
        | expr '+' expr   { $$.value := $1.value + $3.value;   }
        | expr '-' expr   { $$.value := $1.value - $3.value;   }
        | expr '*' expr   { $$.value := $1.value * $3.value;   }
        | expr '/' expr   { $$.value := $1.value DIV $3.value; }
        | expr '%' expr   { $$.value := $1.value MOD $3.value; }
        | '-' expr        { $$.value := - $2.value;            } PREC UNARY
        | LETTER          { $$.value := regs [$1.Scan.value];  }
        | number          { $$.value := $1.value;              }
        .
number  : DIGIT           { $$.value := $1.Scan.value;
             IF $1.Scan.value = 0 THEN base := 8; ELSE base := 10; END; }
        | number DIGIT    { $$.value := base * $1.value + $2.Scan.value; }
        .
```

Appendix 4: Example: Grammar with Predicates and Backtracking

```
GLOBAL {
#include <ctype.h>
#include "Idents.h"

typedef union { tScanAttribute Scan; } tParsAttribute;
}

LOCAL  { char name [256]; }

TOKEN
   IDENTIFIER   = 1
   '('          = 2
   ')'          = 3
   '++'         = 4
   ';'          = 5
   '['          = 6
   ']'          = 7
   '{'          = 8
   '}'          = 9

RULE

compound_statement      : '{' declaration_list statement_list '}'
                        .
declaration_list        :
                        | declaration_list declaration
                        .
statement_list          :
                        | statement_list statement
                        .
declaration             : ? declaration declaration_specifier declarator ';'
                        .
declaration_specifier   : TYPEDEFname [ printf ("trial action\n"); ]
                                      { printf ("final action\n"); }
                        .
declarator              : IDENTIFIER  [ printf ("trial action\n"); ]
                                      { printf ("final action\n"); }
                        | '(' declarator ')'
                        | declarator '[' ']'
                        | compound_statement
                        .
statement               : expression ';'
                        .
expression              : expression '++'
                        | TYPEDEFname '(' expression ')'        /* cast */
                        | IDENTIFIER  '(' expression ')'        /* call */
                        | IDENTIFIER
                        .
TYPEDEFname             : IDENTIFIER$i ?
                          { GetString ($i.Scan.Ident, name), isupper (name [0]) }
                        | IDENTIFIER ? IDENTIFIER
                        .
```

Appendix 5: Example: Tree Construction for MiniLAX in C

```
GLOBAL {
# include "Idents.h"
# include "Tree.h"
   tTree  nInteger, nReal, nBoolean;
   typedef union {
      tScanAttribute Scan;
      tTree          Tree;
   } tParsAttribute;
}
BEGIN {
   BeginScanner ();
   nInteger     = mInteger      ();
   nReal        = mReal         ();
   nBoolean     = mBoolean      ();
}
TOKEN
   Ident        = 1
   IntegerConst = 2
   RealConst    = 3
   PROGRAM      = 4
   ';'          = 5
   'DECLARE'    = 6
   ':'          = 7
   INTEGER      = 8
   REAL         = 9
   BOOLEAN      = 10
   ARRAY        = 11
   '['          = 12
   '..'         = 13
   ']'          = 14
   OF           = 15
   PROCEDURE    = 16
   'BEGIN'      = 17
   '<'          = 18
   '+'          = 19
   '*'          = 20
   NOT          = 21
   '('          = 22
   ')'          = 23
   FALSE        = 24
   TRUE         = 25
   ':='         = 26
   ','          = 27
   IF           = 28
   THEN         = 29
   ELSE         = 30
   'END'        = 31
   WHILE        = 32
   DO           = 33
   READ         = 34
   WRITE        = 35
   VAR          = 36
   '.'          = 37
PREC
   LEFT  '<'
   LEFT  '+'
   LEFT  '*'
   LEFT  NOT
RULE
Prog    : PROGRAM Ident ';' 'DECLARE' Decls 'BEGIN' Stats 'END' '.'
        { TreeRoot = mMiniLax (mProc (mNoDecl (), $2.Scan.Ident.Ident, $2.Scan.Position,
                     mNoFormal (), ReverseTree ($5.Tree), ReverseTree ($7.Tree)));             } .
Decls   : Decl
        { $1.Tree->Decl.Next = mNoDecl (); $$.Tree = $1.Tree;                                 } .
Decls   : Decls ';' Decl
        { $3.Tree->Decl.Next = $1.Tree; $$.Tree = $3.Tree;                                    } .
Decl    : Ident ':' Type
        { $$.Tree = mVar (NoTree, $1.Scan.Ident.Ident, $1.Scan.Position, mRef ($3.Tree));     } .
Decl    : PROCEDURE Ident ';' 'DECLARE' Decls 'BEGIN' Stats 'END'
```

```
        { $$.Tree = mProc (NoTree, $2.Scan.Ident.Ident, $2.Scan.Position, mNoFormal (),
                    ReverseTree ($5.Tree), ReverseTree ($7.Tree));                          } .
Decl    : PROCEDURE Ident '(' Formals ')' ';' 'DECLARE' Decls 'BEGIN' Stats 'END'
        { $$.Tree = mProc (NoTree, $2.Scan.Ident.Ident, $2.Scan.Position, ReverseTree ($4.Tree),
                    ReverseTree ($8.Tree), ReverseTree ($10.Tree));                         } .
Formals : Formal
        { $1.Tree->Formal.Next = mNoFormal (); $$.Tree = $1.Tree;                           } .
Formals : Formals ';' Formal
        { $3.Tree->Formal.Next = $1.Tree; $$.Tree = $3.Tree;                                } .
Formal  : Ident ':' Type
        { $$.Tree = mFormal (NoTree, $1.Scan.Ident.Ident, $1.Scan.Position, mRef ($3.Tree));    } .
Formal  : VAR Ident ':' Type
        { $$.Tree = mFormal (NoTree, $2.Scan.Ident.Ident, $2.Scan.Position, mRef (mRef ($4.Tree)));} .
Type    : INTEGER
        { $$.Tree = nInteger;                                                               } .
Type    : REAL
        { $$.Tree = nReal;                                                                  } .
Type    : BOOLEAN
        { $$.Tree = nBoolean;                                                               } .
Type    : ARRAY '[' IntegerConst '..' IntegerConst ']' OF Type
        { $$.Tree = mArray ($8.Tree, $3.Scan.IntegerConst.Integer, $5.Scan.IntegerConst.Integer,
                    $3.Scan.Position);                                                      } .
Stats   : Stat
        { $1.Tree->Stat.Next = mNoStat (); $$.Tree = $1.Tree;                               } .
Stats   : Stats ';' Stat
        { $3.Tree->Stat.Next = $1.Tree; $$.Tree = $3.Tree;                                  } .
Stat    : Adr ':=' Expr
        { $$.Tree = mAssign (NoTree, $1.Tree, $3.Tree, $2.Scan.Position);                   } .
Stat    : Ident
        { $$.Tree = mCall (NoTree, mNoActual ($1.Scan.Position), $1.Scan.Ident.Ident,
                    $1.Scan.Position);                                                      } .
Stat    : Ident '(' Actuals ')'
        { $$.Tree = mCall (NoTree, ReverseTree ($3.Tree), $1.Scan.Ident.Ident, $1.Scan.Position);} .
Stat    : IF Expr THEN Stats ELSE Stats 'END'
        { $$.Tree = mIf (NoTree, $2.Tree, ReverseTree ($4.Tree), ReverseTree ($6.Tree));    } .
Stat    : WHILE Expr DO Stats 'END'
        { $$.Tree = mWhile (NoTree, $2.Tree, ReverseTree ($4.Tree));                       } .
Stat    : READ '(' Adr ')'
        { $$.Tree = mRead (NoTree, $3.Tree);                                               } .
Stat    : WRITE '(' Expr ')'
        { $$.Tree = mWrite (NoTree, $3.Tree);                                              } .
Actuals : Expr
        { $$.Tree = mActual (mNoActual ($1.Tree->Expr.Pos), $1.Tree);                      } .
Actuals : Actuals ',' Expr
        { $$.Tree = mActual ($1.Tree, $3.Tree);                                            } .
Expr    : Expr '<' Expr
        { $$.Tree = mBinary ($2.Scan.Position, $1.Tree, $3.Tree, Less);                    } .
Expr    : Expr '+' Expr
        { $$.Tree = mBinary ($2.Scan.Position, $1.Tree, $3.Tree, Plus);                    } .
Expr    : Expr '*' Expr
        { $$.Tree = mBinary ($2.Scan.Position, $1.Tree, $3.Tree, Times);                   } .
Expr    : NOT Expr
        { $$.Tree = mUnary ($1.Scan.Position, $2.Tree, Not);                               } .
Expr    : '(' Expr ')'
        { $$.Tree = $2.Tree;                                                               } .
Expr    : IntegerConst
        { $$.Tree = mIntConst ($1.Scan.Position, $1.Scan.IntegerConst.Integer);            } .
Expr    : RealConst
        { $$.Tree = mRealConst ($1.Scan.Position, $1.Scan.RealConst.Real);                 } .
Expr    : FALSE
        { $$.Tree = mBoolConst ($1.Scan.Position, false);                                  } .
Expr    : TRUE
        { $$.Tree = mBoolConst ($1.Scan.Position, true);                                   } .
Expr    : Ident
        { $$.Tree = mIdent ($1.Scan.Position, $1.Scan.Ident.Ident);                        } .
Expr    : Adr '[' Expr ']'
        { $$.Tree = mIndex ($2.Scan.Position, $1.Tree, $3.Tree);                           } .
Adr     : Ident
        { $$.Tree = mIdent ($1.Scan.Position, $1.Scan.Ident.Ident);                        } .
Adr     : Adr '[' Expr ']'
        { $$.Tree = mIndex ($2.Scan.Position, $1.Tree, $3.Tree);                           } .
```

References

[DeP82] F. DeRemer and T. J. Pennello, Efficient Computation of LALR(1) Look-Ahead Sets, *ACM Trans. Prog. Lang. and Systems 4*, 4 (Oct. 1982), 615-649.

[Gro88] J. Grosch, Generators for High-Speed Front-Ends, *LNCS 371*, (Oct. 1988), 81-92, Springer Verlag.

[Gro90] J. Grosch, Lalr - a Generator for Efficient Parsers, *Software—Practice & Experience 20*, 11 (Nov. 1990), 1115-1135.

[GrE90] J. Grosch and H. Emmelmann, A Tool Box for Compiler Construction, *LNCS 477*, (Oct. 1990), 106-116, Springer Verlag.

[GrV] J. Grosch and B. Vielsack, The Parser Generators Lalr and Ell, Cocktail Document No. 8, CoCoLab Germany.

[Groa] J. Grosch, Preprocessors, Cocktail Document No. 24, CoCoLab Germany.

[Grob] J. Grosch, Ast - A Generator for Abstract Syntax Trees, Cocktail Document No. 15, CoCoLab Germany.

[Groc] J. Grosch, Ag - An Attribute Evaluator Generator, Cocktail Document No. 16, CoCoLab Germany.

[Grod] J. Grosch, Reusable Software - A Collection of C-Modules, Cocktail Document No. 30, CoCoLab Germany.

[Groe] J. Grosch, Reusable Software - A Collection of Modula-Modules, Cocktail Document No. 4, CoCoLab Germany.

[Joh75] S. C. Johnson, Yacc — Yet Another Compiler-Compiler, Computer Science Technical Report 32, Bell Telephone Laboratories, Murray Hill, NJ, July 1975.

[KrM81] B. B. Kristensen and O. L. Madsen, Methods for Computing LALR(k) Lookahead, *ACM Trans. Prog. Lang. and Systems 3*, 1 (Jan. 1981), 80-82.

[Mer93] G. H. Merrill, Parsing Non-LR(k) Grammars with Yacc, *Software—Practice & Experience 23*, 8 (Aug. 1993), 829-850.

[Pag77a] D. Pager, The Lane-Tracing Algorithm for Constructing LR(k) Parsers and Ways of Enhancing its Efficiency, *Inf. Sci. 12*, 1 (1977), 19-42.

[Pag77b] D. Pager, A Practical General Method for Constructing LR(k) Parsers, *Acta Inf. 7*, 3 (1977), 249-268.

[PCC85] J. C. H. Park, K. M. Choe and C. H. Chang, A New Analysis of LALR Formalism, *ACM Trans. Prog. Lang. and Systems 7*, 1 (Jan. 1985), 159-175.

[Röh76] J. Röhrich, Syntax-Error Recovery in LR-Parsers, in *Informatik-Fachberichte*, vol. 1, H.-J. Schneider and M. Nagl (ed.), Springer Verlag, Berlin, 1976, 175-184.

[Röh80] J. Röhrich, Methods for the Automatic Construction of Error Correcting Parsers, *Acta Inf. 13*, 2 (1980), 115-139.

[Röh82] J. Röhrich, Behandlung syntaktischer Fehler, *Informatik Spektrum 5*, 3 (1982), 171-184.

Contents