The Parser Generators Lalr and Ell

J. Grosch
B. Vielsack

GESELLSCHAFT FÜR MATHEMATIK
UND DATENVERARBEITUNG MBH

FORSCHUNGSSTELLE FÜR
PROGRAMMSTRUKTUREN
AN DER UNIVERSITÄT KARLSRUHE

Project

Compiler Generation

The Parser Generators Lalr and Ell

Josef Grosch
Bertram Vielsack

July 31, 1992

Report No. 8

Gesellschaft für Mathematik und Datenverarbeitung mbH
Forschungsstelle an der Universität Karlsruhe
Vincenz-Prießnitz-Str. 1
D-7500 Karlsruhe

1. Introduction

This document is the user's manual for the two parser generators *lalr* and *ell* and for the grammar transformation tool *bnf*. All three tools understand one common input language and the two parser generators produce parsers with similar functionality and interfaces. All three tools are described in one manual in order to present the common information only once.

The parser generator *lalr* has been developed with the aim to combine a powerful specification technique for context-free languages with the generation of highly efficient parsers [Gro88, Gro90]. As it processes the class of LALR(1) grammars we chose the name *lalr* to express the power of the specification technique.

The grammars may be written using EBNF constructs. 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 (only synthesized attributes) is provided to allow communication between the semantic actions.

In case of LR-conflicts a derivation tree is printed to ease the location of the problem. The conflict can be resolved by specifying precedence and associativity for terminals and rules. Syntactic errors are handled fully automatically by the generated parsers including error reporting, recovery, and 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 and Modula-2. The generator uses the algorithm described by [DeP82] to compute the look-ahead sets.

Parsers generated by *lalr* are two to three times faster than *Yacc* [Joh75] generated ones. They reach a speed of 580,000 lines per minute on a MC 68020 processor excluding the time for scanning. The size of the parsers is only slightly increased in comparison to *Yacc*, because there is a small price to be paid for the speed.

The parser generator *ell* processes LL(1) grammars which may contain EBNF constructs and semantic actions. It generates recursive descent parsers [Gro88, Gro89b]. A mechanism for L-attribution (inherited and synthesized attributes evaluable during one preorder traversal) is provided. Like *lalr*, syntax errors are handled fully automatic including error reporting from a prototype error module, error recovery, and error repair. *ell* can generate parsers in C and Modula-2. Those satisfied with the restricted power of LL(1) grammars may profit from the high speed of the generated parsers which lies around 900,000 lines per minute.

The tool *bnf* transforms a grammar written in extended BNF into plain BNF. It is used for instance as a preprocessor for *lalr*, because this generator directly understands plain BNF, only.

Besides the input language described in this manual there is a second possibility which can be used to describe grammars for *lalr*. This second possibility is described in the document entitled "Preprocessors" [Gro91a]. The use of the language defined in the current manual works perfectly. However, compared to the second possibility, 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* [Gro91b] and *ag* [Gro89a].
- 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 mechanism uses named attributes instead of the error prone $i construct.

- The attribute grammar is checked for completeness and whether it obeys the SAG property.
- The definition of the type *tParsAttribute* is derived automatically from the attribute declarations.
- Tokens or terminals without attributes might be declared implicitly.
- The grammar and the semantic actions might be separated into several modules.

The rest of this manual is organized as follows: Section 2 gives an overview about the external behaviour of the parser generators. Section 3 explains the common input language of the tools. Section 4 describes the parser generator *lalr*. Section 5 describes the parser generator *ell*. Section 6 describes the transformation tool *bnf*. The Appendices 1 and 2 summarize the syntax of the input language. The Appendices 3 to 5 present examples of parser specifications.

2. Overview

A parser generator transforms a grammar into a parser. The grammar is the specification of a language. The parser is a procedure or a program module for analyzing a given input according to the language specification. The input/output behaviour of the parser generators *lalr* and *ell* is shown in Figure 1. The input is a file that contains the grammar. In case of *lalr* the input may optionally be transformed from extended BNF to plain BNF by the tool *bnf*. The output consists of up to three source modules and a table file. The tools have options to control which outputs should be generated: The module *Parser* contains the desired parsing routine. The module *Errors* is a prototype module to handle syntax error messages. The prototype simply prints the error messages. The program *ParserDrv* is a minimal main program that can serve to test a parser. The file *Parser.Tab* contains data to control the parser. It is generated only if the target language is Modula-2. In the case of *lalr* it contains the parse tables and the case of *ell* it contains information for error recovery.

3. Input Language

The input of a parser generator primarily describes a language. A language is specified conveniently by a grammar. A complete input is divided into the following parts whose order is fixed:

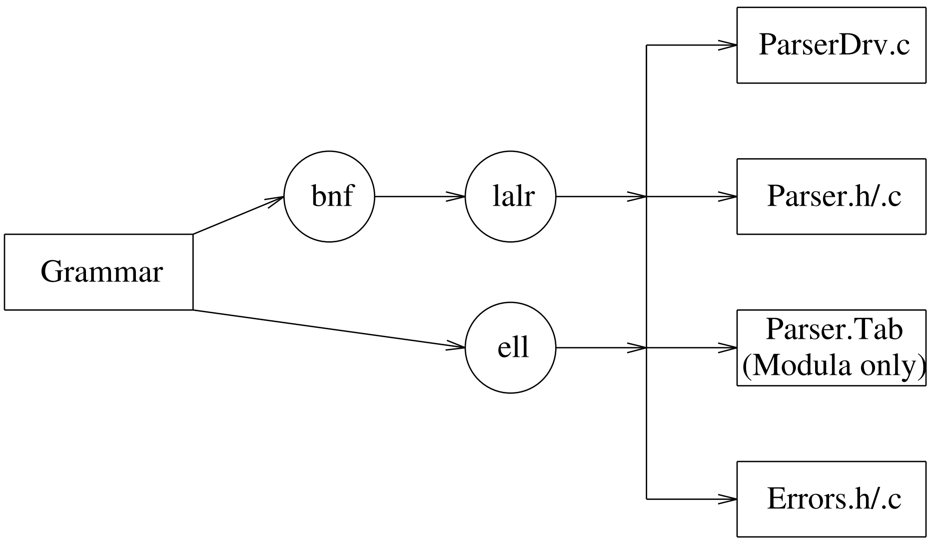


Fig. 1: Input/Output Behaviour of the Parser Generators *lalr* and *ell*

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

The parts specifying the tokens and the grammar are necessary, whereas the other ones are optional. The following sections discuss these parts as well as the lexical conventions. The Appendices 1 and 2 summarize the syntax of the input language using a grammar as well as syntax diagrams.

3.1. Lexical conventions

The input of the parser generators can be written in free format.

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        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.

```
':='   "'"   ''''   "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 in curly brackets '{' and '}'. The characters '{' 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. Within those tokens the escape conventions are disabled and the tokens are left 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"); }
```

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. The first kind of comment is preserved by the grammar transformation tool *bnf*, or in other words, these comments reappear in the output. However, these comments are allowed at certain places of the input, only, as dictated by the syntax of the input language. The second kind of comments may be used anywhere between the lexical elements. They are lost during a transformation using *bnf*.

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

3.2. Names for Scanner and Parser

A grammar may be optionally headed by names for the 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, the parse tables, etc. If the names are missing they default to *Scanner* and *Parser*. In the following we refer to these names by <Scanner> and <Parser>.

3.3. Target Code

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:

EXPORT:	declarations to be included in the interface part.
GLOBAL:	declarations to be included in the implementation part 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.

Example in 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); }
```

3.4. Specification of Terminals

The terminals or tokens of a grammar have to 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.

Example:

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

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

3.5. Precedence and Associativity for Operators

Sometimes grammars are ambiguous and then it is not possible to generate a parser. In many cases ambiguous grammars can be turned into unambiguous ones by the additional specification of precedence and associativity for operators. Operators are the tokens used in expressions. The keyword OPER (for operator) 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:

```
OPER
   NONE  '='
   LEFT  '+' '-'
   LEFT  '*' '/'
   RIGHT '**'
```

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. (See section 4.3. for the use of this information by *lalr*).

3.6. Grammar

The core of a language definition 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. A grammar can be written in extended BNF notation (EBNF). The following example specifies a trivial programming language.

Example:

```
RULE

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

A grammar rule consists of a left-hand side and a right-hand side which are separated by a colon ':'. It is terminated by a dot '.'. The left-hand side has to be a nonterminal which is defined by the right-hand side of 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 conflicts (see section 4.3.). 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 (see section 3.1.). A right-hand side of a grammar rule can be given in extended BNF notation. The following possibilities are available:

A *sequence* of terminals or nonterminals is specified by listing these elements.

```
statement : identifier ':=' expression ';' .
```

Several *alternatives* are separated by bar characters '|'.

```
statement : 'WHILE' expression 'DO' statement ';'
          | 'REPEAT' statement 'UNTIL' expression ';' .
```

Optional parts are enclosed in square brackets '[' and ']'.

```
statement : 'IF' expression 'THEN' statement [ 'ELSE' statement ] ';' .
```

The *repetition* of an element one or more times is expressed by the character '+'.

```
statement : 'BEGIN' statement + 'END' ';' .
```

A *repetition* of an element zero or more times is expressed by the character '*'.

```
statements : statement * .
```

Lists are repetitions where the elements are separated by a delimiter. These lists are characterized by two bar characters '||'. These lists consist of at least one element.

```
identifiers : identifier || ',' .
```

The extended BNF notation is defined more formally as follows:

The rule	abbreviates the rules	
X : u | v .	X : u .	X : v .
X : u [w] v .	X : u Y v .	Y : w | .
X : u w + v .	X : u Y v .	Y : Y w | w .
X : u w * v .	X : u Y v .	Y : Y w | .
X : u w || t v .	X : u Y v .	Y : Y t w | w .

The symbols in the above table have the following meaning:

X	: a nonterminal
Y	: a nonterminal that does not appear elsewhere in the grammar
u, v, w	: arbitrary sequences of terminals or nonterminals
t	: a terminal

The characters used to express extended BNF are treated as some kind of *operators* having different levels of precedence. To change the associativity imposed by the operator precedence, parenthesis '(' and ')' can be used for grouping.

Example:

```
grammar : ( left_hand_side ':' right_hand_side '.' ) + .
```

The following table summarizes the operators and their precedences. The highest precedence is 1 and the lowest is 5. Operators of the same precedence associate from left to right.

Operator	Precedence	Usage
( )	1	grouping
[ ]	1	optional parts
+	2	repetition once or more times
*	2	repetition zero or more times
none	3	sequence
|	4	alternatives
||	5	lists

3.7. Semantic Actions

Semantic actions serve to perform syntax-directed translation. This allows to generate for example an intermediate representation such as a syntax tree or a sequential intermediate language. A semantic action is an arbitrary sequence of statements of the target language enclosed in curly brackets '{' and '}'. One or more semantic actions may be inserted in the right-hand side of a grammar rule.

Example:

```
expression : expression '+' term  { printf ("ADD\n"); } .
```

The generated parser analyzes its input from left to right according to the specified rules. Whenever a semantic action is encountered in a rule the associated 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
```

3.8. Attribute Evaluation

Both parser generators, *lalr* and *ell*, provide a mechanism for the evaluation of attributes during parsing. Attributes are values that are associated with the nonterminal and terminal symbols. The attributes allow to communicate information among grammar rules. Attribute computations are expressed by target code statements with the semantic actions. The syntactic and semantic details of the attribute mechanisms are different for the two parser generators. Therefore they are discussed later in generator specific sections (see sections 4.2. and 5.2.).

3.9. Error Handling

The generated parsers include automatic error recovery, reporting, and repair. There are no instructions necessary to achieve this error handling. The error messages use the terminal symbols of the grammar, only. Therefore self explanatory identifiers or strings are recommended for the denotation of terminals.

4. Lalr

This section describes the use of the LALR(1) parser generator *lalr*.

4.1. Input Language

Basically, *lalr* accepts a language definition as described in section 3. The following peculiarities have to be mentioned:

- *lalr* directly accepts only grammar rules in plain BNF notation. If the grammar uses EBNF constructs such as |, +, *, ||, or [] it has to be converted to plain BNF by the grammar transformer *bnf*. *bnf* can be invoked by providing *lalr* with the option -b.
- The definitions of precedence and associativity for operators and the PREC clause at the end of right-hand sides of rules is recognized by *lalr*. This information is used in order to resolve possible conflicts (see section 4.3.).
- 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. In this case the grammar transformer *bnf* is necessary, again. It transforms the rules by moving all semantic actions to the end of right-hand sides. This is done by the introduction of 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) 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 4.3.).

4.2. S-Attribution

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

For all occurrences of grammar symbols attribute storage areas are maintained. 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 type 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:

```
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;
```

The values of the attributes are computed within the semantic actions. The pseudo variables \$1, \$2, ... 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. If the type *tParsAttribute* is a union or a struct type the pseudo variables may be followed by selectors for members or fields.

Example:

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

The above numbering scheme is valid for semantic actions placed at the end of right-hand sides, only. Actions within a right-hand side may only access attributes of preceding symbols, or in other words, symbols to their left. The indexes start at zero for the immediately preceding symbol and decrease from right to left: \$0, \$-1, \$-2, ... Therefore the attributes of one symbol may be accessed with different indexes depending on the location of the semantic action.

Example:

```
X : a { A } b { B } c { C } .
```

The following table lists for every symbol of the rule the pseudo variable to access its attributes which is different for the semantic actions A, B, and C.

	A	B	C
X	–	–	\$\$
a	\$0	\$−2	\$1
{ A }	\$\$	\$−1	\$2
b	–	\$0	\$3
{ B }	–	\$\$	\$4
c	–	–	\$5
{ C }	–	–	–

4.3. 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 class of LALR(1) grammars. Without additional information ambiguous grammars can not be processed by *lalr*. This section describes how many ambiguity problems can be solved.

The classical example which leads to an ambiguous grammar is the *dangling else* problem. Suppose a grammar contains two rules for IF statements:

```
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.

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 *lalr* would produce a message saying the grammar is not LALR(1). Before we describe what to do in such a case we have to say briefly how the generated parser works.

The generated parser is a stack automaton controlled by a parse table. The automaton is characterized by the contents and the administration of the 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:

Table : State $\times$ Token $\rightarrow$ Action

There are primarily two actions: The action *read* (shift) means to read 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 should contain a read action as well as a reduce action we have a *read-reduce conflict* (or shift-reduce conflict). If a table entry should 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 and the involved state are reported. Furthermore, *lalr* 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 read/shift action receive the precedence and associativity of the token to be read. The rules indicating a reduce action retain their own precedence and associativity. These are either determined by the right-most operator in the rule or an explicitly given PREC clause. The latter dominates any existing operators. If there is at least one rule without precedence and associativity, the conflict is resolved according to the following:

- In case of a read-reduce conflict the read action is prefered.
- In case of a reduce-reduce conflict the rule given first is reduced.

These conflict solutions are reported as warnings. If all involved rules have precedence and associativity values the resolution proceeds as follows:

- In case of a read-reduce conflict and rules with different precedences the action of the rule with highest precedence is prefered. 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, right associativity selects the read action, and no associativity leads to an error message.
- 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 an error message is issued.

  These conflict solutions are reported as informations.

4.4. Conflict Information

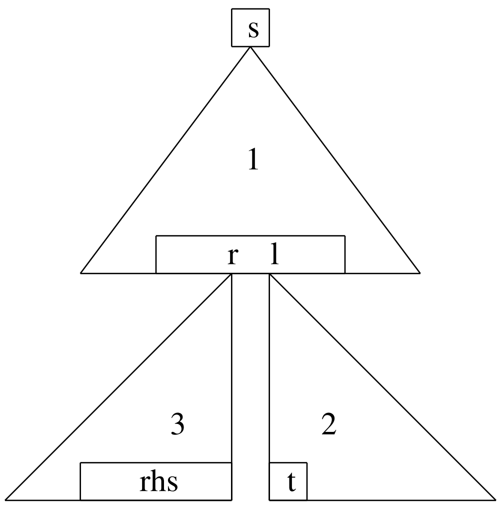
If there are conflicts in the grammar and the option -v (for verbose) of *lalr* is set, then information to ease the location of the reason for the conflicts is produced. This information is written into a file called *_Debug*. For every state with conflicts and for every so-called situation involved in a conflict a derivation tree is printed. 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 mentioned derivation tree explains how a lookahead token and a rule can interfere. The derivation tree has three parts as shown in Figure 2.

The first part describes the derivation from the start symbol of the grammar to an intermediate rule. Two neighbouring symbols 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. Is 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. It describes the derivation of the rule.

This three parts of a tree are printed in an ASCII representation one after the other. The first line contains the start symbol. All following lines contain the right-hand side of a grammar rule. The rules are indented to start below the nonterminal of the left-hand side. To avoid line overflow, dotted edges also refer to the left-hand side nonterminal and allow to shift back to the



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. 2: Structure of the Derivation Tree printed about a Conflict

left margin. The intermediate rule can be recognized as that line where two subtrees start. The left subtree is introduced at least by one "superfluous" colon ':'. In some cases the right subtree consists only of a root symbol. Then it is really only this superfluous colon that marks the intermediate rule. Every derivation tree ends with a possible parser action. The information for every state ends with a summary of the conflict resolution. For every situation it is printed whether it was retained or ignored and for what reason (precedence or associativity).

Example: dangling else

```
State 266

program End-of-Tokens
PROGRAM identifier params ';' block '.'
..............................:
:
labels consts types vars procs BEGIN stmts END
.....................................:
:
stmt
IF expr THEN stmt ELSE stmt
             :
             IF expr THEN stmt
               :
reduce stmt -> IF expr THEN stmt. {ELSE} ?
read   stmt -> IF expr THEN stmt.ELSE stmt ?

ignored  stmt -> IF expr THEN stmt. {ELSE}
retained stmt -> IF expr THEN stmt.ELSE stmt
```

In the above example the first tree part consists of 5 lines (not counting the dotted lines). The symbols *stmt* and ELSE are the roots of the other two tree parts. This location is indicated by the "unnecessary" colon in the following line. After one intermediate line the left subtree derives the conflicting items. The right subtree consists in this case only of the root node (the terminal ELSE) indicating the look-ahead. In general this can be a tree of arbitrary size. The conflict can easily be seen from this tree fragment. If conditional statements are nested as shown, then there is a read reduce conflict.

4.5. 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 3 gives an overview of the modules and their interface objects. Circles denote procedures, squares denote variables, and arrows represent procedure calls or variable access. As the details of these interfaces depend on the implementation language they are discussed in language specific sections.

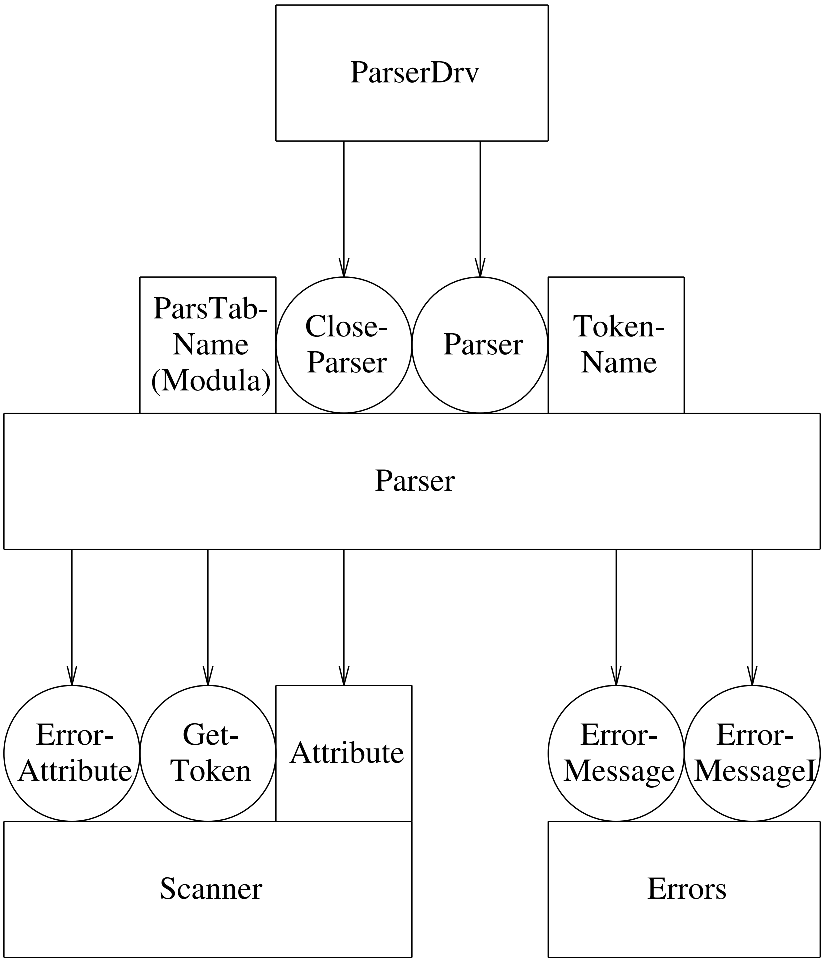


Fig. 3: Interface Objects of the Modules

4.5.1. C

4.5.1.1. Parser Interface

The parser interface in the file <Parser>.h has the following contents:

```
extern  char *  <Parser>_TokenName [];
extern  int     <Parser>           ();
extern  void    Close<Parser>      ();
```

- The procedure <Parser> is the generated parsing procedure. It returns the number of syntax errors. A return value of zero indicates a syntactically correct input.
- The contents of the target code section named BEGIN is put into a procedure called Begin<Parser>. This procedure is called automatically upon the first invocation of the procedure <Parser>.
- 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.
- 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. It is used for example by the standard error handling module to provide expressive messages.

4.5.1.2. 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 items:

```
# include "Positions.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 ();
```

- 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.
- Additional properties 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 a record type like <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 by the procedure <Scanner>_GetToken. In case of syntax errors this field is passed as parameter 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 of tokens are transferred from the global variable <Scanner>_Attribute to this field.
- During automatic error repair a parser may insert tokens. In this case the parser calls the procedure <Scanner>_ErrorAttribute to ask for the additional properties of an inserted token which is given by the parameter Token. The procedure should return in the second argument called Attribute a default value for the additional properties of this token.

4.5.1.3. 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 any arbitrary way. The parser generator can provide a prototype error handling module in the files Errors.h and 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 items:

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

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

# define xxString               7       /* info classes         */

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

- There are four messages a generated parser may report. They are encoded by the first group of constant definitions above. 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 restart points.
- The procedure ErrorMessageI is like the procedure ErrorMessage with additional Information. The latter is characterized by a class or type indication and an (untyped) pointer. Two types of additional information are used by the parser. During error repair tokens might be inserted. These are reported one by one and are classified as xxString (char *). At every syntax error the set of legal or expected tokens is reported using the classification xxString, too.

4.5.1.4. Parser Driver

To test a generated parser a main program is necessary. 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"

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

4.5.2. Modula-2

4.5.2.1. Parser Interface

The parser interface in the file <Parser>.md has the following contents:

```
DEFINITION MODULE <Parser>;

VAR       ParsTabName : ARRAY [0..128] OF CHAR;

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

END <Parser>.
```

- The procedure <Parser> is the generated parsing procedure. It returns the number of syntax errors. A return value of zero indicates a syntactically correct input.
- The array ParsTabName specifies the name of the file containing the parser tables. It is initialized with the string "<Parser>.Tab". Therefore, the parser tables are read by default

from a file with this name in the current directory. If a different name or location is desired an arbitrary path name can be assigned to this array before calling <Parser> the first time.

- The contents of the target code section named BEGIN is put into a procedure called Begin<Parser>. This procedure is called automatically upon the first invocation of the procedure <Parser>.
- 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.
- 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. It is used for example by the standard error handling module to provide expressive messages.

4.5.2.2. Scanner Interface

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

```
DEFINITION MODULE <Scanner>;

IMPORT Positions;

TYPE      tScanAttribute = RECORD Position: Positions.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.
- Additional properties 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 a record type like 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 by the procedure GetToken. In case of syntax errors this field is passed as parameter 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 of tokens are transferred from the global variable Attribute to this field.
- During automatic error repair a parser may insert tokens. In this case the parser calls the procedure ErrorAttribute to ask for the additional properties of an inserted token which is given by the parameter Token. The procedure should return in the second argument called pAttribute a default value for the additional properties of this token.

4.5.2.3. 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 any arbitrary way. The parser generator can provide a prototype error handling module in the files Errors.md and Errors.mi whose procedures immediately print the information passed as arguments.

```
DEFINITION MODULE Errors;

FROM SYSTEM     IMPORT ADDRESS;
FROM Positions  IMPORT tPosition;

CONST
   SyntaxError          = 1     ;       (* error codes          *)
   ExpectedTokens       = 2     ;
   RestartPoint         = 3     ;
   TokenInserted        = 4     ;
   WrongParseTable      = 5     ;
   OpenParseTable       = 6     ;
   ReadParseTable       = 7     ;

   Fatal                = 1     ;       (* error classes        *)
   Error                = 3     ;
   Repair               = 5     ;
   Information          = 7     ;

   Integer              = 1     ;       (* info classes         *)
   String               = 7     ;
   Array                = 8     ;

PROCEDURE ErrorMessage  (ErrorCode, ErrorClass: CARDINAL; Position: tPosition);
PROCEDURE ErrorMessageI (ErrorCode, ErrorClass: CARDINAL; Position: tPosition;
                         InfoClass: CARDINAL; Info: ADDRESS);

END Errors.
```

- There are seven messages a generated parser may report. They are encoded by the first group of constant definitions above. 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, restart points, and problems encountered during reading of the parse tables.
- The procedure ErrorMessageI is like the procedure ErrorMessage with additional Information. The latter is characterized by a class or type indication and an (untyped) pointer. Two types of additional information are used by the parser. During error repair tokens might be inserted. These are reported one by one and 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).

4.5.2.4. Parser Driver

To test a generated parser a main program is necessary. 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.
```

4.6. Error Recovery

The generated parsers include information and algorithms to handle syntax errors completely automatically. *lalr* uses the complete backtrack-free method described by [Röh76, Röh80, Röh82] and provides expressive reporting, recovery, and repair. Every incorrect input is "virtually" transformed into a syntactically correct program with the consequence of only executing a "correct" sequence of semantic actions. Therefore the following compiler phases like semantic analysis don't have to bother with syntax errors. *lalr* provides a prototype error module which prints messages as shown in the following:

Example: Automatic Error Messages

Source Program:

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

Error Messages:

```
3, 13: Error       syntax error
3, 13: Information expected symbols: ')' '*' '+' '-' '/' '<' '<=' '=' '<>'
                                     '>' '>=' AND DIV IN MOD OR
3, 15: Information restart point
3, 15: Repair      symbol inserted : ')'
3, 15: Repair      symbol inserted : THEN
```

Internally the error recovery works as follows:

- The location of the syntax error is reported.
- All the tokens that would be a legal continuation of the program are computed and reported.
- All the tokens that can serve to continue parsing are computed. A minimal sequence of tokens is skipped until one of these tokens is found.
- The recovery location is reported.
- Parsing continues in the so-called repair mode. In this mode the parser behaves as usual except that no tokens are read from the input. Instead a minimal sequence of tokens is synthesized to repair the error. The parser stays in this mode until the input token 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.

4.7. Usage

NAME

lalr – LALR(1) parser generator

SYNOPSIS

lalr [-c|-m] [-b][-d][-e][-h][-l][-p][-s][-g][-v] [-cs][n] <file>

DESCRIPTION

Lalr is a parser generator for highly efficient parsers which processes the class of LALR(1) grammar. The grammars may be written using EBNF constructs. 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 (only synthesized attributes) is provided to allow communication between the semantic actions. Ambiguities in the grammar may be solved by specifying precedence and associativity for tokens and grammar rules.

In case of LR-conflicts a derivation tree is printed to ease the location of the problem. The conflict can be resolved by specifying precedence and associativity for terminals and rules. Syntactic errors are handled fully automatically by the generated parsers including error reporting, recovery, and repair. The generated parsers are table-driven.

The generated parser needs a scanner, an error handler, and a few modules from a library of reusable modules. A primitive scanner can be requested with the option -s. The option -e produces a prototype error handler. Errors detected during the analysis of the grammar are reported on standard error. If the generator finds LR-conflicts and option -v is given the file _Debug will be produced. This file will give detailed informations about the conflicts. If any conflict has been repaired using precedence and associativity a conflict description is written to the file _Debug, too.

OPTIONS

- c generate C source code
- m generate Modula-2 source code (default)
- a generate all = -d -e -p -s
- b run the preprocessor bnf and feed its output into lalr
- d generate definition module
- e generate module for error handling
- p generate parser driver
- s generate mini scanner
- g generate # line directives
- v verbose: produce debugging information in file _Debug
- cs reduce the number of case labels in switch or case statements by mapping so-called read-reduce to reduce states (increases run time a little bit but decreases code size, might be necessary due to compiler restrictions)
- <n> generate switch or case statements with at most n case labels (might be necessary due to compiler restrictions)
- h print further help information
- l print complete (error) listing

FILES

_Debug	file containing the debug information if grammar is not LALR(1) and option -v is given

if output is in C:

<Parser>.h	specification of the generated parser
<Parser>.c	body of the generated parser
<Parser>Drv.c	body of the parser driver
Errors.h	specification of error handler
Errors.c	body of error handler
<Scanner>.h	specification of scanner
<Scanner>.c	body of scanner

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
<Parser>.Tab	tables to control the generated parser
Errors.md	definition module of error handler
Errors.mi	implementation module of error handler
<Scanner>.md	definition module of scanner
<Scanner>.mi	implementation module of scanner

SEE ALSO

J. Grosch, B. Vielsack: "The Parser Generators Lalr and Ell", GMD Forschungsstelle an der Universitaet Karlsruhe, Compiler Generation Report No. 8, 1991

J. Grosch: "Lalr - a Generator for Efficient Parsers", Software - Practice & Experience, 20 (11), 1115-1135, Nov. 1990

5. Ell

This section describes the use of the LL(1) parser generator *ell*.

5.1. Input Language

Basically, *ell* accepts a language definition as described in section 3. The following peculiarities have to be mentioned:

- A grammar may optionally be headed by names for the 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, the parse tables, etc. If the names are missing they default to *Scanner* and *Parser*. In the following we refer to these names by <Scanner> and <Parser>.

- A grammar rule may optionally contain local target code:

  ```
  Rule : Identifier ':' 'LOCAL' Action RightSide '.'
  ```

  A rule is transformed into a procedure. The local target code is placed at the beginning of this procedure. The code may contain declarations and statements (C only). This feature is in effect in addition to the target code section LOCAL specified at global level (at the beginning of a grammar). The latter target code section is inserted in every procedure preceding the rule specific target code.

- Definitions of precedence and associativity are ignored.
- *ell* directly processes grammars written in EBNF notation.
- In contrast to *lalr*, semantic actions may be inserted freely at any places within rules without causing conflicts.

5.2. L-Attribution

According to [Wil79] an attribute grammar which can be evaluated during LL(1)-parsing is called an L-attributed grammar. The notion L-attribution means that all attributes can be evaluated in a single top-down left-to-right tree walk.

ell distinguishes three kinds of grammar symbols: nonterminals, terminals, and literals. Literals are similar to terminals and are denoted by strings. Terminals and nonterminals are denoted by identifiers. Terminals and nonterminals can be associated with arbitrary many attributes of arbitrary types. The computation of the attribute values takes place in the semantic action parts of a rule. The attributes are accessed by an attribute designator which consists of the name of the grammar symbol, a dot character, and the name of the attribute. For the target language C the dot character has to be replaced by the symbol '->' whenever attributes of the left-hand side are accessed. The reason is that left-hand side attributes are output parameters and therefore the formal parameter is of a pointer type. As several grammar symbols with the same name can occur within a rule, the grammar symbols are denoted unambiguously by appending numbers to their names. The left-hand side symbol always receives the number zero. For every (outermost) alternative of the right-hand side, the symbols with the same name are counted starting from one.

Example in C: Evaluation of simple arithmetic expressions

```
expr    : ( [ '+' ] term        { expr0->value =  term1.value; }
          | '-' term            { expr0->value = -term2.value; }
          )
          ( '+' term            { expr0->value += term3.value; }
          | '-' term            { expr0->value -= term4.value; }
          ) *
        .
term    : fact                  { term0->value =  fact1.value; }
          ( '*' fact            { term0->value *= fact2.value; }
          | '/' fact            { term0->value /= fact3.value; }
          ) *
        .
fact    : const                 { fact0->value = const1.value; }
        | '(' expr ')'          { fact0->value =  expr1.value; }
        .
```

Example in Modula-2: Evaluation of simple arithmetic expressions

```
expr    : ( [ '+' ] term { expr0.value :=   term1.value;                 }
          | '-' term     { expr0.value := - term2.value;                 }
          )
          ( '+' term     { INC (expr0.value, term3.value);               }
          | '-' term     { DEC (expr0.value, term4.value);               }
          ) *
        .
term    : fact           { term0.value := fact1.value;                   }
          ( '*' fact     { term0.value := term0.value * fact2.value;     }
          | '/' fact     { term0.value := term0.value DIV fact3.value;   }
          ) *
        .
fact    : const          { fact0.value := const1.value;                  }
        | '(' expr ')'   { fact0.value := expr1.value;                   }
        .
```

Two types are used to describe attributes. The type *tScanAttribute* describes the attributes of terminals. It is exported from the scanner. The type *tParsAttribute* describes the attributes of nonterminals. It has to be declared by the user in the EXPORT target code section. Usually this type is a union or variant record type with one member or variant for every nonterminal that has attributes. Every member or variant may be described by a struct or record type if a nonterminal has several attributes. The attributes of terminals are automatically transferred from the scanner to the parser by accessing the external variable *Attribute* that is exported by the scanner.

Example in C:

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

Example in Modula-2:

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

5.3. Non LL(1) Grammars

Sometimes grammars do not obey the LL(1) property. They are said to contain LL(1) conflicts. A well-known example is the dangling-else problem of Pascal: in case of nested it-then-else statements it may not be clear to which IF an ELSE belongs (see section 4.3.). It is very easy to solve these conflicts in hand-written solutions. *ell* handles LL(1) conflicts in the following ways:

- Several alternatives (operator |) cause a conflict if their FIRST sets are not disjoint: the alternative given first is selected.
- An optional part (operators [] and *) causes a conflict if its FIRST set is not disjoint from its FOLLOW set: the optional part will be analyzed because otherwise it would be useless.
- Parts that may be repeated at least once cause a conflict if their FIRST and FOLLOW sets are not disjoint (as above): the repetition will be continued because otherwise it would be executed only once.

With the above rules it can happen that alternatives are never taken or that it is impossible for a repetition to terminate for any correct input. These cases as well as left recursion are considered to be serious design faults in the grammar and are reported as errors. Otherwise LL(1) conflicts are resolved as described above and reported as warnings.

5.4. 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 3 gives an overview of the modules and their interface objects. As the details of these interfaces depend on the implementation language they are discussed in language specific sections.

5.4.1. C

5.4.1.1. Parser Interface

The parser interface in the file <Parser>.h has the following contents:

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

typedef ...                    <Parser>_tParsAttribute;

extern <Parser>_tParsAttribute <Parser>_ParsAttribute;
extern char *                  <Parser>_TokenName [];
extern int                     <Parser>          ();
extern void                    Close<Parser>     ();
```

- The procedure <Parser> is the generated parsing procedure. It returns the number of syntax errors. A return value of zero indicates a syntactically correct input.
- The variable <Parser>_ParsAttribute of type <Parser>_tParsAttribute holds the attribute values of the root symbol of the grammar. If the root symbol has inherited attributes these have to be assigned to this variable before calling the procedure <Parser>.
- The contents of the target code section named BEGIN is put into a procedure called Begin<Parser>. This procedure is called automatically upon the first invocation of the procedure <Parser>.
- 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.
- 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. It is used for example by the standard error handling module to provide expressive messages.

5.4.1.2. Scanner Interface

A generated parser needs the following objects usually provided by a scanner module:

```
# include "Positions.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 ();
```

- 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.
- Additional properties 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 a record type like <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 by the procedure <Scanner>_GetToken. In case of syntax errors this field is passed as parameter to the error handling routines.
- During automatic error repair a parser may insert tokens. In this case the parser calls the procedure <Scanner>_ErrorAttribute to ask for the additional properties of an inserted token which is given by the parameter Token. The procedure should return in the second argument called Attribute a default value for the additional properties of this token.

5.4.1.3. 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 any arbitrary way. The parser generator can provide a prototype error handling module in the files Errors.h and 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 items:

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

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

# define xxString               7       /* info classes         */

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

- There are four messages a generated parser may report. They are encoded by the first group of constant definitions above. 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 restart points.
- The procedure ErrorMessageI is like the procedure ErrorMessage with additional Information. The latter is characterized by a class or type indication and an (untyped) pointer. Only the type String (Char *) is used by the parser to classify the additional information. During error repair tokens might be inserted. These are reported one by one and are classified as String (char *). At every syntax error the set of legal or expected tokens is reported using the classification String, too.

5.4.1.4. Parser Driver

To test a generated parser a main program is necessary. 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"

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

5.4.2. Modula-2

5.4.2.1. Parser Interface

The parser interface in the file <Parser>.md has the following contents:

```
DEFINITION MODULE <Parser>;

TYPE tParsAttribute     = ...

VAR ParsAttribute       : tParsAttribute;
VAR ParsTabName         : ARRAY [0..128] OF CHAR;

PROCEDURE <Parser>      (): INTEGER;
PROCEDURE Close<Parser> ();
PROCEDURE xxTokenName   (Token: SHORTCARD; VAR Name: ARRAY OF CHAR);

END <Parser>.
```

- The procedure <Parser> is the generated parsing procedure. It returns the number of syntax errors. A return value of zero indicates a syntactically correct input.
- The variable ParsAttribute of type tParsAttribute holds the attribute values of the root symbol of the grammar. If the root symbol has inherited attributes these have to be assigned to this variable before calling the procedure <Parser>.
- The array ParsTabName specifies the name of the file containing the parser tables. It is initialized with the string "<Parser>.Tab". Therefore, the parser tables are read by default from a file with this name in the current directory. If a different name or location is desired an arbitrary path name can be assigned to this array before calling <Parser> the first time.
- The contents of the target code section named BEGIN is put into a procedure called Begin<Parser>. This procedure is called automatically upon the first invocation of the procedure <Parser>.

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

5.4.2.2. Scanner Interface

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

```
DEFINITION MODULE <Scanner>;

IMPORT Positions;

TYPE      tScanAttribute = RECORD Position: Positions.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.
- Additional properties 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 a record type like 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 by the procedure GetToken. In case of syntax errors this field is passed as parameter to the error handling routines.
- During automatic error repair a parser may insert tokens. In this case the parser calls the procedure ErrorAttribute to ask for the additional properties of an inserted token which is given by the parameter Token. The procedure should return in the second argument called Attribute a default value for the additional properties of this token.

5.4.2.3. 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 any arbitrary way. The parser generator can provide a prototype error handling module in the files Errors.md and Errors.mi whose procedures immediately print the information passed as arguments.

```
DEFINITION MODULE Errors;

FROM SYSTEM     IMPORT ADDRESS;
FROM Positions  IMPORT tPosition;

CONST
   SyntaxError          = 1     ;       (* error codes          *)
   ExpectedTokens       = 2     ;
   RestartPoint         = 3     ;
   TokenInserted        = 4     ;
   ReadParseTable       = 7     ;

   Fatal                = 1     ;       (* error classes        *)
   Error                = 3     ;
   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);

END Errors.
```

- There are fife messages a generated parser may report. They are encoded by the first group of constant definitions above. 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, restart points, and problems encountered during reading of the parse tables.
- The procedure ErrorMessageI is like the procedure ErrorMessage with additional Information. The latter is characterized by a class or type indication and an (untyped) pointer. Two types of additional information are used by the parser. During error repair tokens might be inserted. These are reported one by one and 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).

5.4.2.4. Parser Driver

To test a generated parser a main program is necessary. 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.
```

5.5. Error Recovery

The generated parsers include information and program code to handle syntax errors completely automatically and provide expressive error reporting, recovery, and repair. Every incorrect input is "virtually" transformed into a syntactically correct program with the consequence of executing only a "correct" sequence of semantic actions. Therefore the following compiler phases like semantic analysis don't have to bother with syntax errors. *ell* provides a prototype error module which prints messages as shown in the following:

Example: Automatic Error Messages

Source Program:

```
MODULE test;
BEGIN
   IF (a = ] 1 write (a) END;
END test.
```

Error Messages:

```
3, 12: Error       syntax error
3, 12: information expected symbols: Ident Integer Real String '(' '+' '-' '{' 'NOT'
3, 14: Information restart point
3, 16: Error       syntax error
3, 16: Information restart point
3, 16: Repair      symbol inserted : ')'
3, 16: Repair      symbol inserted : 'THEN'
```

Internally the error recovery works as follows:

- The location of the syntax error is reported.
- If possible, the tokens that would be a legal continuation of the program are reported.
- The tokens that can serve to continue parsing are computed. A minimal sequence of tokens is skipped until one of these tokens is found.
- The recovery location (restart point) is reported.
- Parsing continues in the so-called repair mode. In this mode the parser behaves as usual except that no tokens are read from the input. Instead a minimal sequence of tokens is synthesized to repair the error. The parser stays in this mode until the input token can be accepted. The synthesized tokens are reported as inserted symbols. 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.

5.6. Usage

NAME

ell – recursive descent parser generator

SYNOPSIS

ell [-options] [file]

DESCRIPTION

The parser generator *Ell* processes LL(1) grammars which may contain EBNF constructs and semantic actions. It generates recursive descent parsers. A mechanism for L-attribution (inherited and synthesized attributes evaluable during one preorder traversal) is provided. Syntax errors are handled fully automatic including error reporting from a prototype error module, error recovery, and error repair.

The grammar is either read from the file given as argument or from standard input. The output is written to the files <Parser>.md and <Parser>.mi (Modula-2) or <Parser>.h and <Parser>.c (C). Errors detected during the analysis of the grammar are reported on standard error.

The generated parser needs a few additional modules:
First, a scanner (<Scanner>.md/<Scanner>.c, <Scanner>.mi/<Scanner>.h) containing the function GetToken () and the global variable Attribute. A very primitive scanner can be requested with the option -s.
Second, a main program. Option -p will provide a simple parser driver (<Parser>Drv.mi/<Parser>Drv.c).
Third, an error handling module called Errors has to provide the procedures ErrorMessage and ErrorMessageI. A prototype error handler can be requested with the option -e .

OPTIONS

- c generate C code
- d generate definition part
- e generate prototype error handler
- g generate # line directives
- h provide help information
- i generate implementation part
- m generate Modula-2 code (default)
- p generate parser driver
- s generate (simple) scanner

FILES

if output is in C:

<Parser>.h	specification of the generated parser
<Parser>.c	body of the generated parser
<Parser>Drv.c	body of the parser driver
Errors.h	specification of error handler
Errors.c	body of error handler
<Scanner>.h	specification of scanner
<Scanner>.c	body of scanner

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
Errors.md	definition module of error handler
Errors.mi	implementation module of error handler
<Scanner>.md	definition module of scanner
<Scanner>.mi	implementation module of scanner
<Parser>.Tab	table to control error recovery

SEE ALSO

J. Grosch, B. Vielsack: "The Parser Generators Lalr and Ell", GMD Forschungsstelle an der Universitaet Karlsruhe, Compiler Generation Report No. 8, 1991

J. Grosch: "Efficient and Comfortable Error Recovery in Recursive Descent Parsers", Structured Programming, 11, 129-140 (1990)

6. Bnf

The grammar transformer *bnf* converts a grammar written in extended BNF (EBNF) into an equivalent grammar in plain BNF. In the plain BNF grammar semantic actions appear at the end of rules, only. The conversion from EBNF to BNF is performed according to the following:

```
EBNF                BNF
X : u | v .         X : u .             X : v .
X : u [ w ] v .     X : u Y v .         Y : .          Y : w .
X : u w + v .       X : u Y v .         Y : Z .        Y : Y Z .        Z : w .
X : u w * v .       X : u Y v .         Y : .          Y : Y w .
X : u w || t v .    X : u Z Y v .       Y : .          Y : Y t Z .      Z : w .
X : u ( w ) v .     X : u Y v .         Y : w .
X : u { A } v .     X : u Y v .         Y : { A } .
```

6.1. Usage

NAME

bnf – convert a grammar from EBNF to BNF

SYNOPSIS

bnf [-c|-m] [-l][-g] <file>

DESCRIPTION

Bnf translates a context-free grammar in EBNF into an equivalent grammar in BNF, which is written to standard output. The result can be used as input for the parser generator lalr.

OPTIONS

c the target language is C

m the target language is Modula-2 (default)

l produce a long error listing

g generate # line directives

SEE ALSO

J. Grosch, B. Vielsack: "The Parser Generators Lalr and Ell", GMD Forschungsstelle an der Universitaet Karlsruhe, Compiler Generation Report No. 8, 1991

Acknowledgements

J. Grosch programmed the table interpreter and the error recovery of *lalr* and designed the generated code and the error recovery of *ell*. B. Vielsack programmed the generator *lalr* and the transformer *bnf*. D. Kuske programmed the generator *ell*. B. Vielsack added to the latter the generation of C code, the L-attribution mechanism, and the disambiguating rules for non-LL(1) grammars. The first version of this manual was written by B. Vielsack. This second version of the manual reuses some parts of the first version.

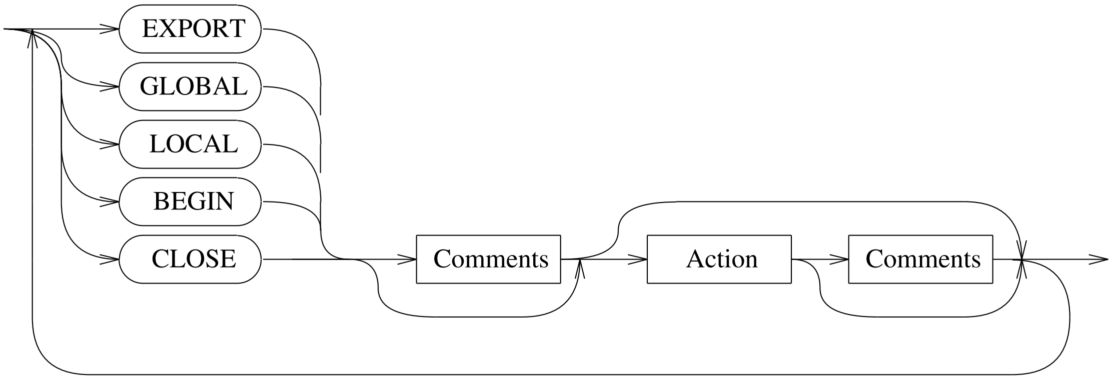
Appendix 1: Syntax of the Input Language

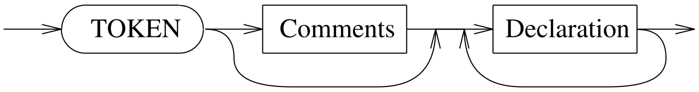
```
RULE

Grammar         : CommentPart Names Decl Tokens Oper RuleList
                .
Names           : ScannerName ParserName
                .
ScannerName     :
                | 'SCANNER'
                | 'SCANNER' Identifier
                .
ParserName      :
                | 'PARSER'
                | 'PARSER' Identifier
                .
Decl            : Decl 'EXPORT' CommentPart Actions
                | Decl 'GLOBAL' CommentPart Actions
                | Decl 'LOCAL'  CommentPart Actions
                | Decl 'BEGIN'  CommentPart Actions
                | Decl 'CLOSE'  CommentPart Actions
                |
                .
Actions         : Action CommentPart
                |
                .
Tokens          : 'TOKEN' CommentPart Declarations
                .
Declarations    : Declarations Declaration
                | Declaration
                .
Declaration     : Terminal Coding CommentPart
                .
Coding          : '=' Number
                |
                .
Oper            : 'OPER' CommentPart Precedences
                |
                .
Precedences     : Precedence Precedences
                |
                .
Precedence      : Associativity Operators CommentPart
                .
Associativity   : 'LEFT'
                | 'RIGHT'
                | 'NONE'
                .
Operators       : Operator Operators
                | Operator
                .
Operator        : Terminal
                .
Terminal        : Identifier
                | String
                .
RuleList        : 'RULE' CommentPart Rules
                .
Rules           : Rules Rule
                | Rule
                .
```

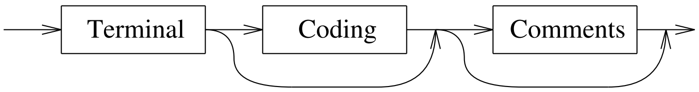
```
Rule            : Identifier ':' LocalCode RightSide '.' CommentPart
                .
LocalCode       : 'LOCAL' Action                   /* ell only */
                |
                .
RightSide       : Expressions PrecPart '|' RightSide
                | Expressions PrecPart
                .
PrecPart        : 'PREC' Terminal
                |
                .
Expressions     : Expression Expressions
                |
                .
Expression      : Unit
                | Unit '*'
                | Unit '+'
                | Unit '||' Unit
                .
Unit            : '[' Alternative ']'
                | '(' Alternative ')'
                | Identifier
                | String
                | Action
                .
Alternative     : Expressions '|' Alternative
                | Expressions
                .
CommentPart     : CommentPart Comment
                |
                .
                                        /* lexical grammar */
Identifier      : Letter
                | `_`
                | `\`
                | Identifier Letter
                | Identifier Digit
                | Identifier '_'
                .
Number          : Digit
                | Number Digit
                .
String          : "'" Characters "'"
                | '"' Characters '"'
                .
Action          : '{' Characters '}'
                .
Comment         : '(*' Characters '*)'
                .
Comment2        : '/*' Characters '*/'
                .
Characters      :
                | Characters Character
                .
```

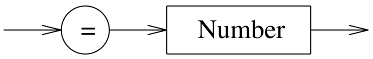
Appendix 2: Syntax Diagrams

Grammar

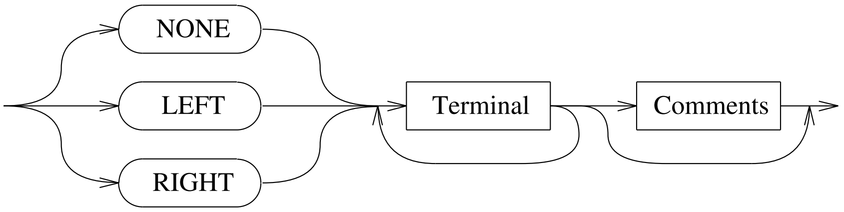

Decl


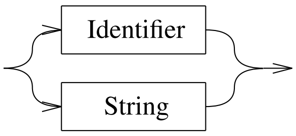
Tokens


Declaration


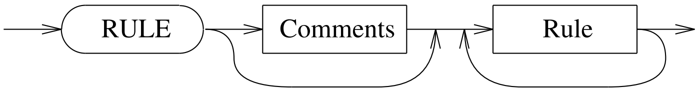
Coding


Oper


Precedence


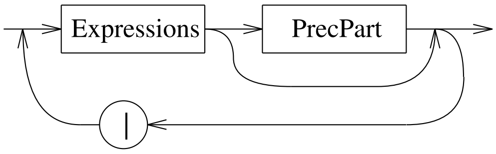
Terminal


Rules

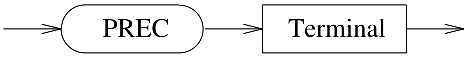


Rule

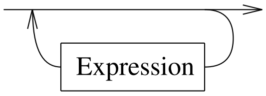
RightSide



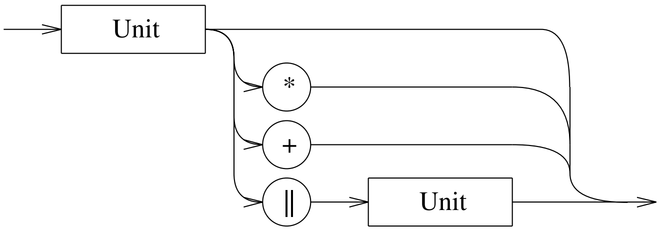
PrecPart



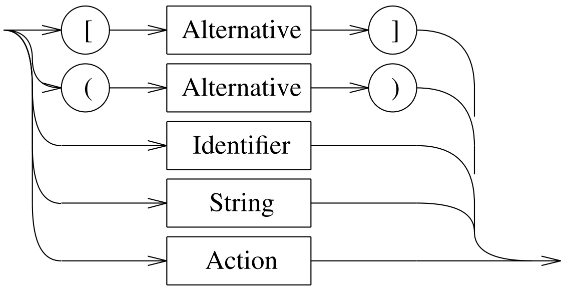
Expressions



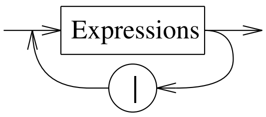
Expression



Unit

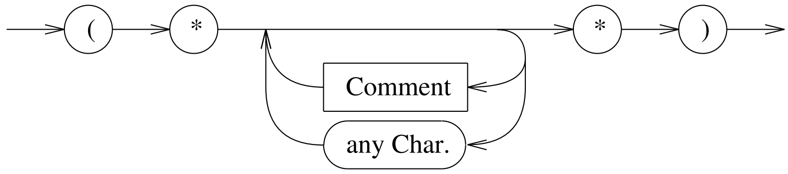


Alternative



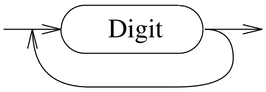
Comments

Comment

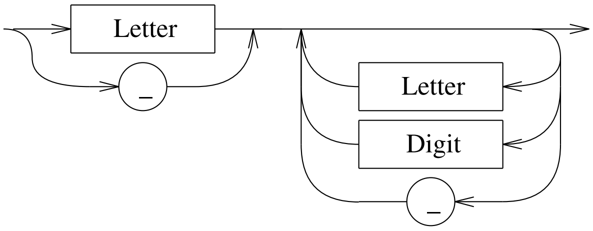


any Char.: all characters except of the character sequences '(*' and '*)' are allowed

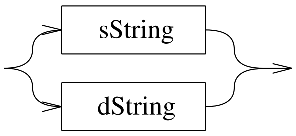
Number



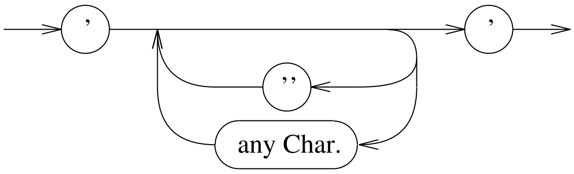
Identifier



String

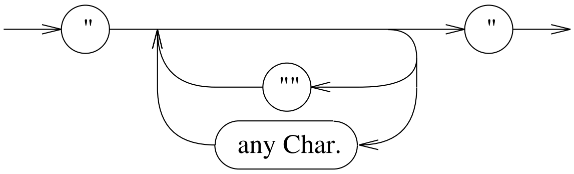


sString



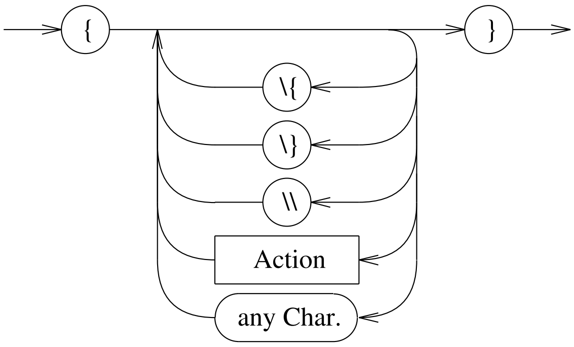
any Char.: all characters except of the single quote and the new line character are allowed

dString

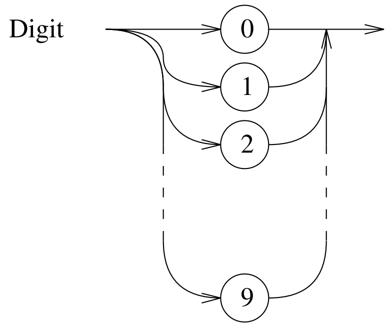


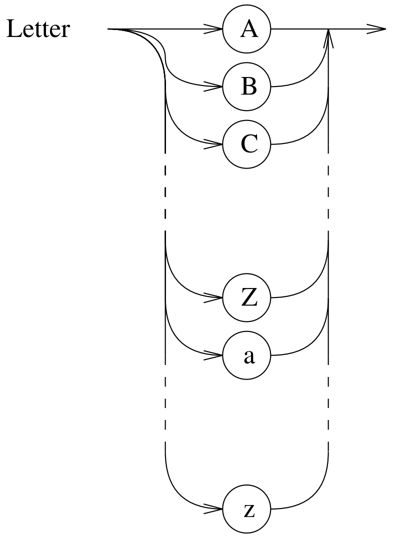
any Char.: all characters except of the double quote and the new line character are allowed

Action

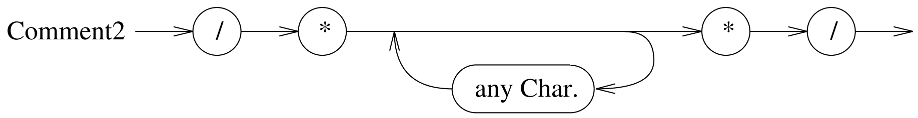


any Char.: all characters except of '\', '{', and '}' are allowed





This second kind of comment is allowed anywhere in the input.



any Char.: all characters except of the character sequence '*/' are allowed

Appendix 3: Example: Desk Calculator for Lalr (BNF, Modula-2)

```
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

OPER
        LEFT    '+'     '-'
        LEFT    '*'     '/'     '%'
        LEFT    UMINUS

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         { $$.value := - $2.value;            } PREC UMINUS
        | LETTER           { $$.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: Desk Calculator for Ell (EBNF, C)

```
EXPORT  { typedef struct { int value; } tParsAttribute; }

BEGIN   { BeginScanner (); }

TOKEN

   const        = 1
   '('          = 2
   ')'          = 3
   '+'          = 4
   '-'          = 5
   '*'          = 6
   '/'          = 7
   'NL'         = 8

RULE

list    : ( expr 'NL'           { printf ("%d\n", expr1.value); } ) *
        .
expr    : ( [ '+' ] term        { expr0->value =  term1.value; }
          | '-' term            { expr0->value = -term2.value; }
          )
          ( '+' term            { expr0->value += term3.value; }
          | '-' term            { expr0->value -= term4.value; }
          ) *
        .
term    : fact                  { term0->value =  fact1.value; }
          ( '*' fact            { term0->value *= fact2.value; }
          | '/' fact            { term0->value /= fact3.value; }
          ) *
        .
fact    : '(' expr ')'          { fact0->value =  expr1.value; }
        | const                 { fact0->value = const1.value; }
        .
```

Appendix 5: Example: Tree Construction for MiniLAX (BNF, 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

OPER
   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. 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.

[Gro89a] J. Grosch, Ag - An Attribute Evaluator Generator, Compiler Generation Report No. 16, GMD Forschungsstelle an der Universität Karlsruhe, Aug. 1989.

[Gro89b] J. Grosch, Efficient and Comfortable Error Recovery in Recursive Descent Parsers, Compiler Generation Report No. 19, GMD Forschungsstelle an der Universität Karlsruhe, Dec. 1989.

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

[Gro91a] J. Grosch, Preprocessors, Compiler Generation Report No. 24, GMD Forschungsstelle an der Universität Karlsruhe, Feb. 1991.

[Gro91b] J. Grosch, Ast - A Generator for Abstract Syntax Trees, Compiler Generation Report No. 15, GMD Forschungsstelle an der Universität Karlsruhe, Sep. 1991.

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

[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.

[Wil79] R. Wilhelm, Attributierte Grammatiken, *Informatik Spektrum 2*, 3 (1979), 123-130.

Contents