The Parser Generator Ell

J. Grosch

DR. JOSEF GROSCH

COCOLAB - DATENVERARBEITUNG

GERMANY

Cocktail

Toolbox for Compiler Construction

The Parser Generator Ell

Josef Grosch

April 17, 1998

Document No. 8

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

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

1. Introduction

This document is the user’s manual for the parser generator *ell* and for the grammar transformation tool *bnf*. The two tools understand one common input language. Both tools are described in one manual in order to present the common information only once.

The parser generator *ell* processes LL(1) grammars which may contain EBNF constructs and semantic actions. It generates recursive descent parsers [Gro88, Gro]. 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. *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 on a MC 68020 processor.

The tool *bnf* transforms a grammar written in extended BNF into plain BNF. It can be used for instance as a preprocessor for the parser generator *lark*, because this generator directly understands plain BNF and simple EBNF, only.

The rest of this manual is organized as follows: Section 2 gives an overview about the external behaviour of the parser generator. Section 3 explains the common input language of the tools. Section 4 describes the parser generator *ell*. Section 5 describes the transformation tool *bnf*. The Appendices 1 and 2 summarize the syntax of the input language. Appendix 3 presents an example of a parser specifications using an EBNF grammar.

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 generator *ell* is shown in Figure 1. The input is a file that contains the grammar. The output consists of up to three source modules. 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.

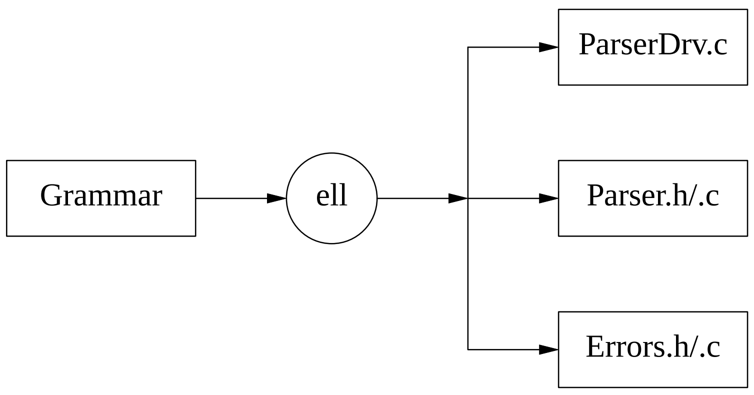


Fig. 1: Input/Output Behaviour of the Parser Generator *ell*

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:

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       START         TOKEN
```

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

```
1   27
```

A string is a sequence of characters enclosed either in single quotes "’" or double quotes ’"’. If the delimiting quote character is to be included within the string it has to be written twice. Strings denote terminal symbols or tokens.

```
’:=’   "’"   ’’’’   "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"); }
```

The parser generators do not know about the syntax of the target language except for strings. Strings are checked for correct syntax in statements as well as in comments, because the tool does not distinguish between between statements and comments. This has the advantage that strings are copied unchanged to the generated output. However, it has the disadvantage that single and double quotes have to appear in pairs in comment lines contained in semantic actions. Unpaired quotes would be reported as incorrect strings.

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

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

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 Z + v .	X : u Y v .	Y : Y Z | Z .
X : u Z * v .	X : u Y v .	Y : Y Z | .
X : u Z || t v .	X : u Y v .	Y : Y t Z | Z .

The symbols in the above table have the following meaning:

X	: a nonterminal
Y	: a nonterminal that does not appear elsewhere in the grammar
Z	: a terminal, nonterminal, semantic action, or expression in parentheses ()
u, v, w	: sequences of terminals, nonterminals, and semantic actions
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

The parser generator, *ell* provides 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 mechanism are discussed later in section 4.2.

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. Ell

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

4.1. 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, 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 parser generators such as e. g. *lark*, semantic actions may be inserted freely at any places within rules without causing conflicts.

4.2. 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;
```

4.3. 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.

4.4. 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.

4.4.1. C

4.4.1.1. 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.

4.4.1.2. Scanner Interface

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

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

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

4.4.1.3. 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 kinds of messages a generated parser may report. They are encoded by the first group of constant definitions above. The messages are classified with respect to severity according to the second group of constant definitions.
- The procedure ErrorMessage is used by the parser to report a message, its class, and its source position. It is used for syntax errors and 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.

4.4.1.4. 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;
}
```

4.4.2. Modula-2

4.4.2.1. Parser Interface

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

```
DEFINITION MODULE <Parser>;

TYPE tParsAttribute     = ...

VAR ParsAttribute       : tParsAttribute;

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 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.

4.4.2.2. Scanner Interface

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

```
DEFINITION MODULE <Scanner>;

IMPORT Position;

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

END <Scanner>.
```

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

4.4.2.3. 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 Position   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 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 Array (ARRAY OF CHAR). At every syntax error the set of legal or expected tokens is reported using the classification String (tString).

4.4.2.4. 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.
```

4.5. 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.

4.6. 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) providing the function GetToken () and the global variable Attribute.
Second, a main program that calls the generated parsing routine. Option -p will provide a simple parser driver (<Parser>Drv.mi/<Parser>Drv.c).
Third, an error handling module called Errors providing the procedures ErrorMessage and ErrorMessageI.

OPTIONS

c	generate C source code
d	generate header file or definition module
-f[*prefix*]	generate constant declarations for tokens in header file using prefix (default: t_)
g	generate # line directives
h	print help information
i	generate implementation part or module
j	treat undeclared symbols as terminal symbols
-l*dir*	specify the directory dir where ell finds its data files
m	generate Modula-2 code (default)
p	generate main program to be used as test driver

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

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

SEE ALSO

J. Grosch: "The Parser Generator Ell", CoCoLab Germany, Document No. 8

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

J. Grosch: "Efficient and Comfortable Error Recovery in Recursive Descent Parsers", CoCo-Lab Germany, Document No. 19

5. 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 S-attribution mechanism defined for *lark* can not be converted by *bnf*. If such an attribution is necessary it is recommended to use plain BNF directly. The conversion from EBNF to BNF is performed according to the following rules:

```
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 } .
```

5.1. Usage

NAME

bnf − convert a grammar from EBNF to BNF

SYNOPSIS

bnf [-c|-m|-eiffel] [-h][-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 lark.

OPTIONS

c the target language is C

m the target language is Modula-2 (default)

eiffel
the target language is Eiffel

h print further help information

l print complete (error) listing

g generate # line directives

SEE ALSO

J. Grosch: "The Parser Generator Ell", CoCoLab Germany, Document No. 8

Acknowledgements

J. Grosch designed the generated code and the error recovery of *ell*. B. Vielsack programmed 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.

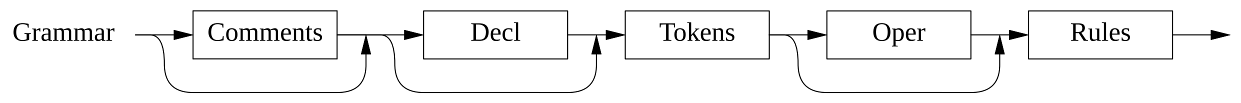
Appendix 1: Syntax of the Input Language

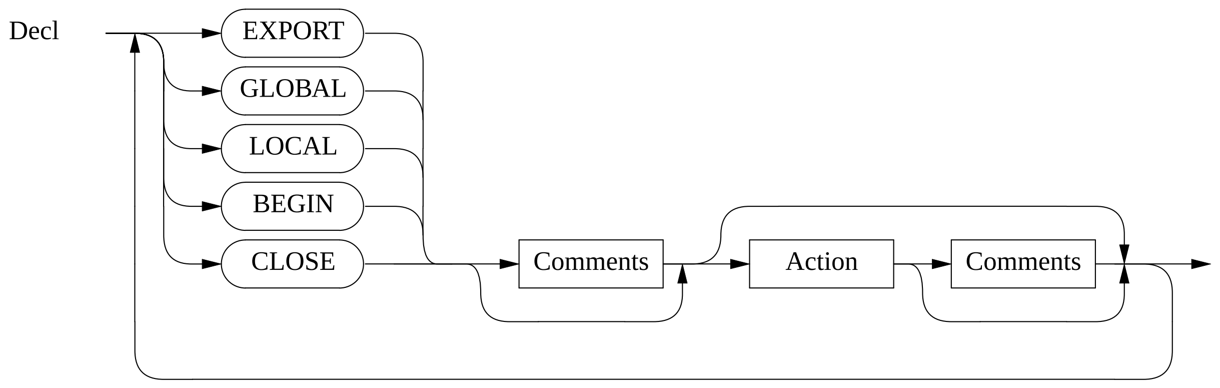
```
RULE

Grammar         : CommentPart Names Decl Tokens Oper StartPart Rules
                .
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
                .
StartPart       :
```

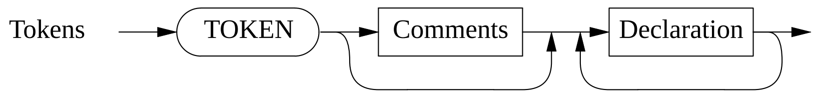
```
                | 'START' Nonterminals
                .
Nonterminals    :
                | Nonterminals Identifier Comma_opt
                .
Comma_opt       :
                | ','
                .
RuleList        : 'RULE' CommentPart Rules
                .
Rules           : Rules Rule
                | 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
                |
                .
Identifier      : Letter                           /* lexical grammar */
                | '_'
                | '\'
                | Identifier Letter
                | Identifier Digit
                | Identifier '_'
                .
Number          : Digit
                | Number Digit
                .
String          : "'" Characters "'"
                | '"' Characters '"'
                .
```

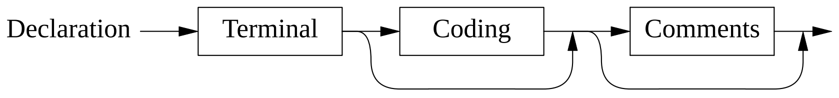
```
Action          : '{' Characters '}'
                .
Comment         : '(*' Characters '*)'
                .
Comment2        : '/*' Characters '*/'
                .
Characters      :
                | Characters Character
                .
```

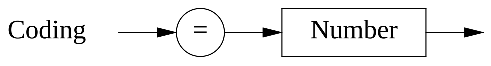
Appendix 2: Syntax Diagrams

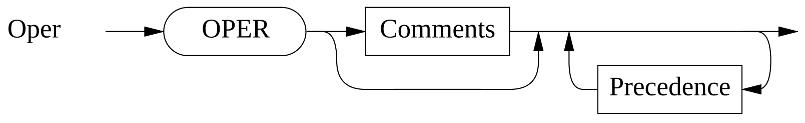


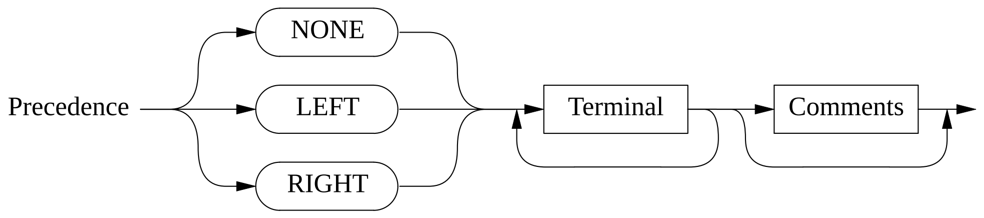


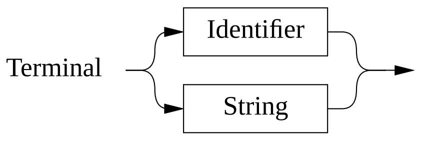


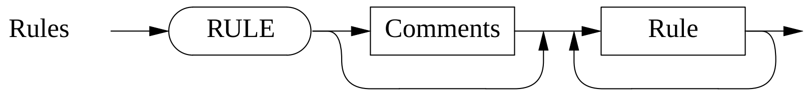


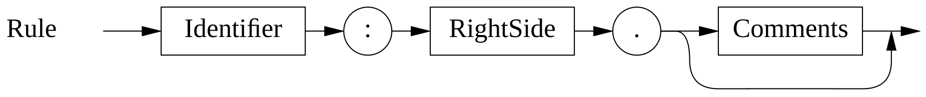


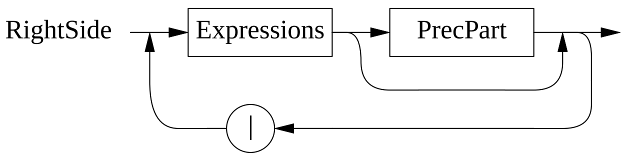


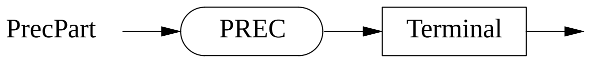


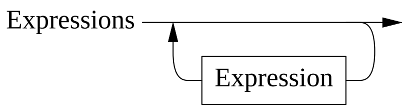

Terminal
Identifier
String

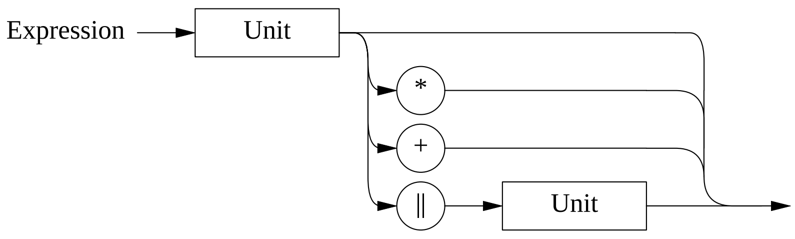

Rules
RULE
Comments
Rule

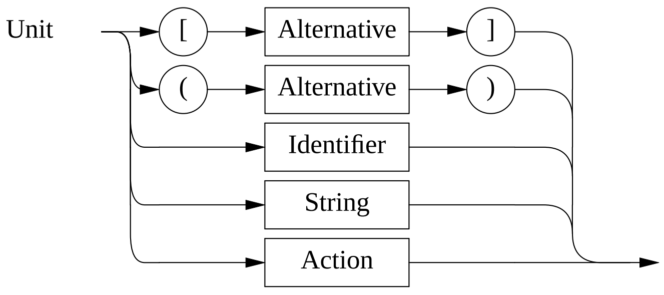

Rule
Identifier
:
RightSide
.
Comments

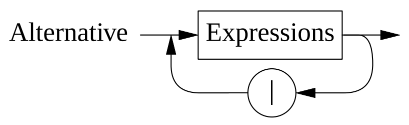

RightSide
Expressions
PrecPart
|

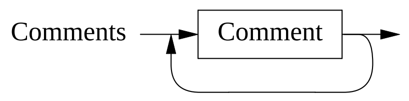

PrecPart
PREC
Terminal

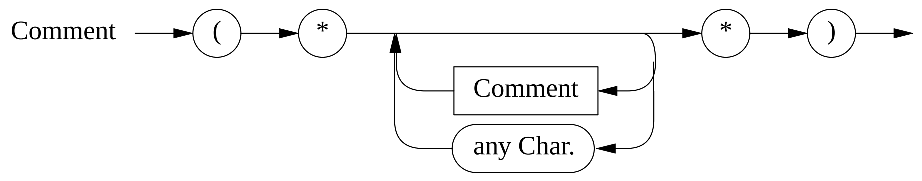

Expressions
Expression


Expression
Unit
*
+
||
Unit

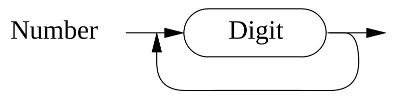

Unit
[
Alternative
]
(
Alternative
)
Identifier
String
Action

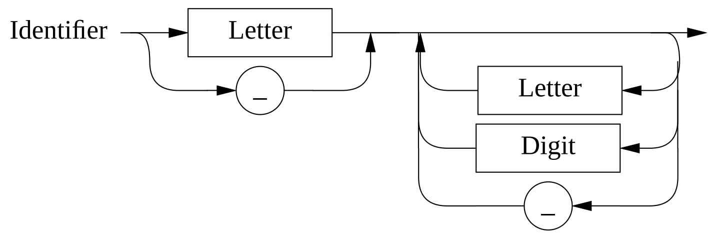


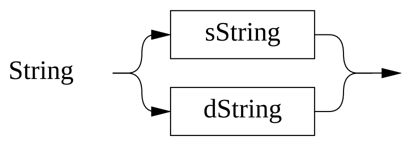


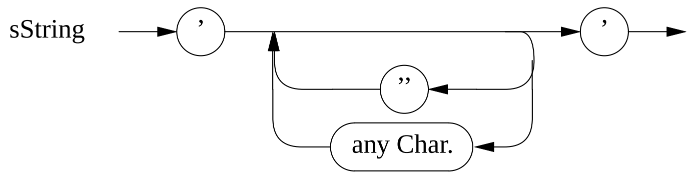


any Char.: all characters except of the character sequences ’(*’ and ’*)’ are allowed

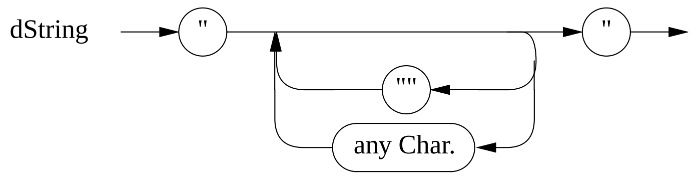




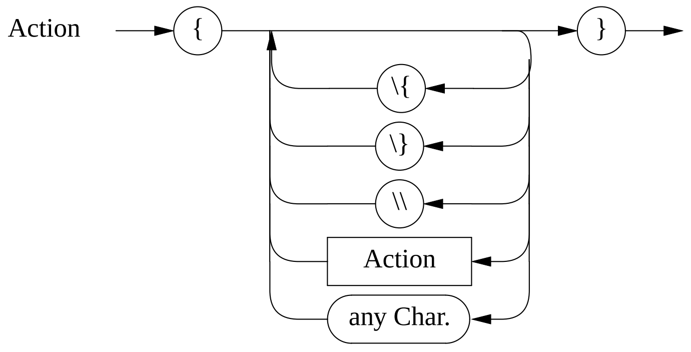




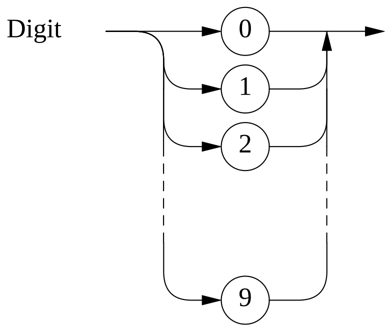
any Char.: all characters except of the single quote and the new line character are allowed

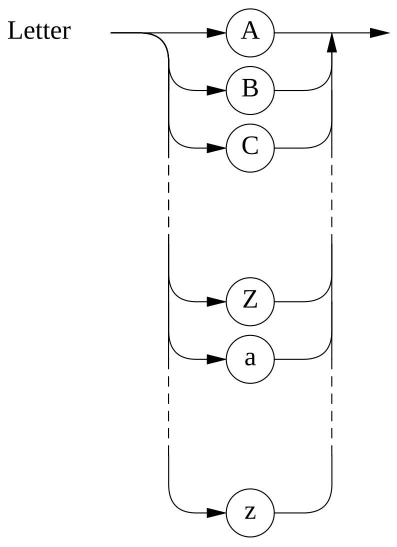


any Char.: all characters except of the double quote and the new line character are allowed

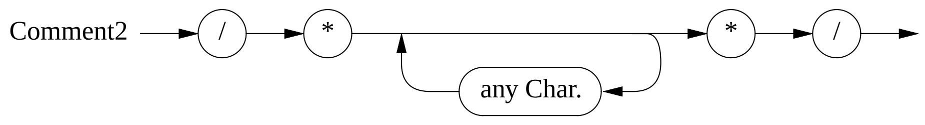


any Char.: all characters except of ’\’, ’{’, and ’}’ are allowed





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 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; }
        .
```

References

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

[Gro] J. Grosch, Efficient and Comfortable Error Recovery in Recursive Descent Parsers, Cocktail Document No. 19, CoCoLab Germany.

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

Contents