Rex
A Scanner Generator

J. Grosch

GESELLSCHAFT FÜR MATHEMATIK
UND DATENVERARBEITUNG MBH

FORSCHUNGSSTELLE FÜR
PROGRAMMSTRUKTUREN
AN DER UNIVERSITÄT KARLSRUHE

Project

Compiler Generation

Rex - A Scanner Generator

Josef Grosch

July 31, 1992

Report No. 5

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

1. Introduction

Rex generates programs to be used in lexical analysis of text. A typical application is the generation of scanners for compilers. Rex stands for Regular EXpression tool. In principle it is a remake of LEX [Les75].

Rex processes a specification containing regular expressions to be searched for, and actions written in C or Modula-2 to be executed when expressions are found. Unrecognized portions of the input are copied by default to standard output. Rex generates a table-driven scanner consisting of a scanner routine and control tables. The scanner routine implements a tunnel automaton [Gro89] and contains a copy of the specified actions.

The scanners generated by Rex are 5 times faster and up to 5 times smaller than those generated by LEX. It is possible to reach a speed of 180,000 to 195,000 lines per minute on a MC 68020 processor (including input from file). If hashing of identifiers is performed additionally the speed is between 125,000 and 150,000 lines per minute. The generator Rex itself is 10 to 20 times faster than LEX in typical cases. Like LEX, Rex has all the features necessary to scan contemporary languages: that is the left and the right context can be taken into account to identify a token. The left context is handled by so-called start states and the right context by additional regular expressions. The source coordinates (line and column number) of recognized words are calculated automatically. Scanners can be generated in the languages C and Modula-2. Rex itself is implemented in Modula-2.

The following chapters constitute the user manual of Rex. Chapter 2 gives an overview of the operation of Rex and how its output is to be integrated in e. g. compilers. Chapter 3 describes the specification language. Chapter 4 summarizes the predefined items of the specification language. Chapter 5 contains the specification of the interface of the generated scanners. Chapter 6 shows how to invoke and use Rex. Chapter 7 contains some details of the implementation. Chapter 8 describes the differences between Rex and LEX for those already familiar with LEX. The appendices contain a grammar for the input language and some examples.

2. Overview

Figure 1 gives an overview of the observable behaviour of Rex. It takes as input a specification of a lexical analyser written in the language described in the next chapter. The output is the source text of a scanner and scanner tables (in case of Modula-2). The source text consists of a specification and a body part. These parts are files with the suffixes 'h' and 'c' if C is the target language. In the case of Modula-2 the parts are a definition and an implementation module. The scanner requires a source module to get blocks of characters e. g. by input from file. Rex can be asked to provide a prototype source module which performs input from the UNIX standard input file. Additionally Rex can be asked to provide a main program to serve as test driver of the scanner. This main program calls the scanner routine until the end of the input is reached.

The above mentioned source programs constitute the minimum configuration to run the generated scanner. What is happening after the compilation of the program modules is shown in the "run time" half of Fig. 1. During initialization the scanner reads its tables from a file (in case of Modula-2 – C uses initialized arrays). Then the scanner driver starts calling the scanner routine which in turn sometimes calls the source module routines to get characters. The data flow is in the opposite direction. The source module returns blocks of characters to the scanner. The scanner analyzes the character stream, executes the associated actions upon finding character sequences matched by regular expressions, and eventually returns tokens to the scanner driver. In general the scanner driver can be replaced by any other main program or subroutine like e. g. a parser.

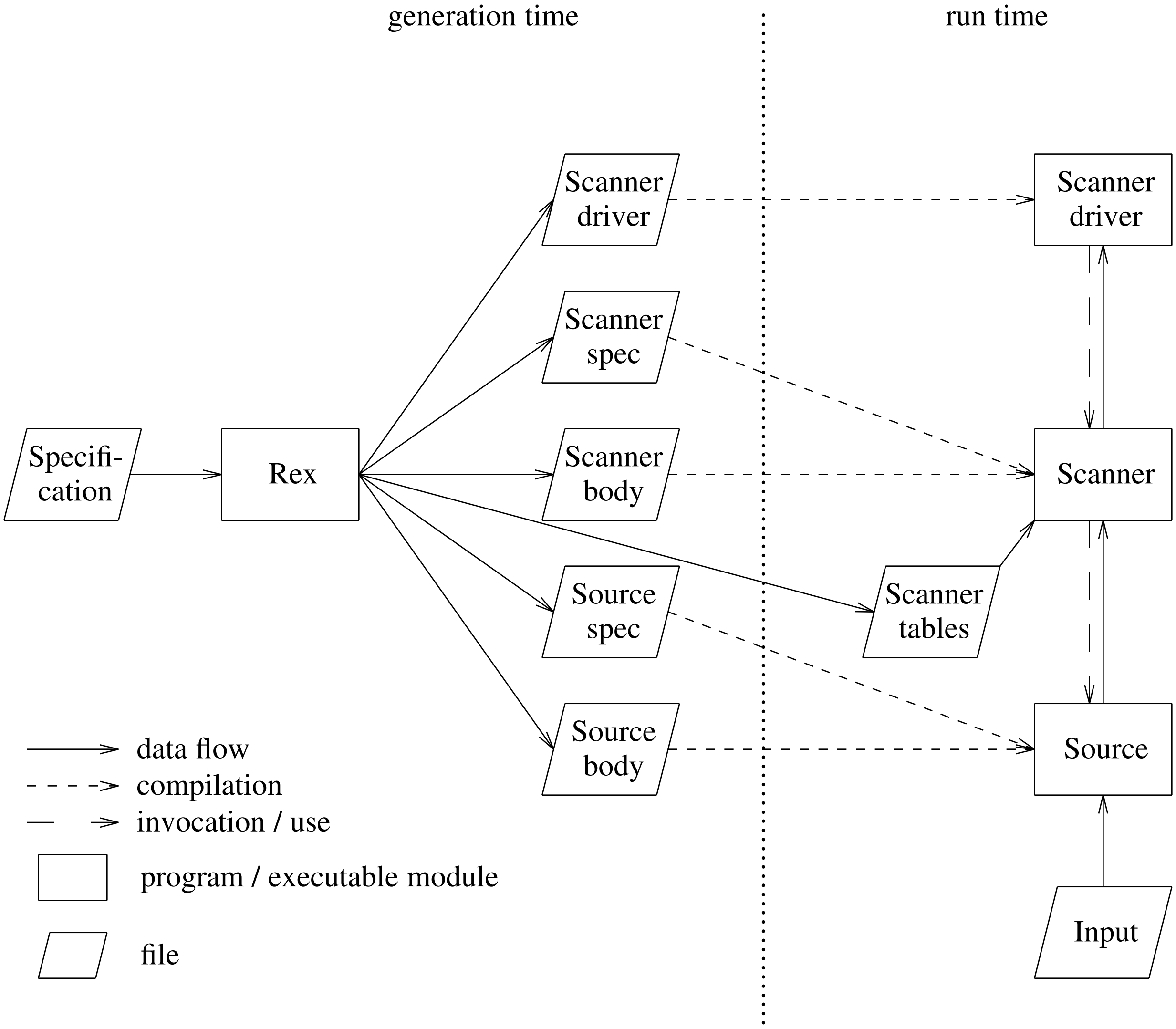

Fig. 1: Rex Overview

3. Specification Language

The input of Rex consists of 3 parts:

- code written in the target language to be copied unchanged to the output (see 3.7.)
- definitions of named regular expressions and start states (see 3.4. + 3.5.)
- a set of regular expressions with associated actions written in the target language (see 3.2.)

The first two parts are optional. We discuss the three parts in reverse order after introducing some lexical conventions.

3.1. Lexical Conventions

The specification can be written in unformatted manner. That means white space in the form of blanks, tab characters, and newline characters has no meaning except to separate other items. Comments are written in the style of C: text included in '/*' and '*/' is ignored. Comments may not be nested. The specification uses a few keywords which should be escaped if needed as identifiers (see below):

```
BEGIN          CLOSE          DEFAULT        DEFINE         EOF
EXPORT         GLOBAL         LOCAL          NOT            RULE
RULES          SCANNER        START
```

The following special characters are used as operators, delimiters, or escape characters:

```
=  .  ,  :  :-  "  #  +  -  *  /  |  ?  (  )  [  ]  {  }  <  >  \
```

Besides keywords and the above special characters a scanner specification is composed of characters, numbers, identifiers, strings, and actions.

A character denotes itself. Special characters have to be escaped using a preceding escape character. The escape character is a backslash: '\'. For certain non-graphic characters the same escape sequences as in C are possible:

newline	NL	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f

Other unprintable characters are represented by the escape character followed by an integer decimal number giving the internal coding.

```
;  \+  \\  \n  \10
```

Numbers denote numerical integer values. They consist of a sequence of digits.

```
8  12  0
```

Identifiers are used to refer to named entities. They consist of a letter followed by letters, digits, or underscore characters '_'. Lower case as well as upper case letters are possible. If an identifier is not defined its character sequence is treated as a string. Identifiers that are keywords have to be escaped by a preceding escape character.

```
letter  HexDigit  under_score  \BEGIN  END
```

Strings denote a sequence of characters. They consist of a sequence of characters enclosed in double quotes '"'. It is not possible to include a double quote or an newline character into a string. No escape is needed within strings. It is a shorthand for escaping a whole sequence of characters.

```
"BEGIN"  ":="  "\"
```

Actions are statements to be copied unchanged into the generated code. The statements have to be written in the desired target language. The actions have to be enclosed in braces '{' '}'. 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. There are additionally statements available to aid in scanning (see section 4.4.).

```
{ printf ("BEGIN recognized\n"); }
{ return SymBegin; }
{ if (level > 0) { GetWord (String); Concatenate (Word, String); } }
{ printf ("} recognized\n"); }
```

3.2. Regular Expressions

In general the specification of a scanner consists of the keyword RULE or RULES followed by a list of regular expressions each one associated with an action.

```
RULE
BEGIN    : { printf ("BEGIN recognized"); }
END      : { printf ("END   recognized"); }
;        : { printf (";     recognized"); }
```

The scanner generated from the above example specification would print an appropriate message upon finding one of the character sequences 'BEGIN', 'END', or ';' in the input whenever they appear. We say a character sequence and a regular expression match if the character sequence has a structure according to the regular expression.

In general the input of the scanner is searched for character sequences which match one of the specified regular expressions and the associated action is executed. Input characters which are not matched by any regular expression are copied by default to standard output.

The syntax to write regular expressions is as follows (see Appendix 1 for a complete definition of the syntax). The productions are given in increasing precedence:

```
Reg_Expr: Reg_Expr '|' Reg_Expr
        | Reg_Expr Reg_Expr
        | Reg_Expr '+'
        | Reg_Expr '*'
        | Reg_Expr '?'
        | Reg_Expr '[' Number ']'
        | Reg_Expr '[' Number '-' Number ']'
        | '(' Reg_Expr ')'
        | Character_Set
        | Character
        | Identifier
        | String
        .
```

- A character is matched by a single identical character.

```
a                   matches the character 'a'
\t                  matches a tab character
\n                  matches a newline character
\10                 matches a newline character (only if ASCII is used)
\\                  matches the character '\'
```

- A string is matched by a character sequence identical to the characters that make up the string.

```
":="                matches the character sequence ':='
"\"                 matches the character '\'
```

- An identifier may be defined to refer to a regular expression. In this case it matches the same characters as the regular expression. An undefined identifier is treated like a string, it matches its own character sequence.

```
END                 matches the character sequence 'END'
\NOT                matches the character sequence 'NOT'
```

- A number is treated like a string, it matches its own character sequence.

```
007                 matches the character sequence '007'
```

- A character set matches one arbitrary character contained in the set. It is written as a sequence of characters enclosed in braces. Ranges may be used to include intervals of characters. The same escapes as described for characters may be used. Unprintable characters and the following ones have to be escaped within character sets:

```
'-'  '}'  ' '  '\'
```

The predefined identifier ANY stands for a character set containing every character except the newline character. If a character set is preceded by the operator '−' it matches one arbitrary character except the ones contained in the set.

```
{ +\-*/ }           matches the arithmetic operators + - * /
{ A-Z a-z 0-9 }     matches all letters and digits
- { \n }            matches all characters except the newline character
ANY                 matches all characters except the newline character
```

- Two regular expressions separated by the operator '|' match characters that are matched by the first or by the second regular expressions.

```
a | b               matches the characters 'a' or 'b'
```

- Two regular expressions following each other with no operator in between match the concatenation of character sequences matched by the single regular expressions.

```
a b                 matches the character sequence 'ab'
```

- The operator '?' matches a character sequence matched by the preceding regular expression or the empty character sequence. In other words, the specified characters are optional.

```
a b ?               matches the character sequences 'a' and 'ab'
```

- The operator '+' matches a character sequence which can be matched by the repetition of the preceding regular expression 1 or more times.

```
a +                 matches the character sequences 'a', 'aa', 'aaa', ...
```

- The operator '*' matches a character sequence which can be matched by the repetition of the preceding regular expression zero or more times.

```
a b *               matches the character sequences 'a', 'ab', 'abb',
                    'abbb', ...
```

- A regular expression followed by a number in brackets matches a character sequence which can be matched by the repetition of the preceding regular expression exactly the times specified by the number.

```
a [4]               matches the character sequence 'aaaa'
```

- A regular expression followed by a range in brackets matches a character sequence which can be matched by the repetition of the preceding regular expression a number of times lying in between of the two given numbers.

```
a [2-4]             matches the character sequences 'aa', 'aaa', and 'aaaa'
```

- Parentheses '(' ')' can be used for grouping in more complex regular expressions.

```
(a | b+)? (c d)*    matches strings like 'acdcd', 'cdcdcd', 'bcd', or 'bbb';
                    but not 'ab', 'abb', or 'abcd'.
```

A complete regular expression which is not part of any other regular expression is called a pattern. A pattern is matched exactly in the same way as regular expressions. It can be augmented by the following specifications.

- A pattern preceded by the operator '<' matches a character sequence only if it appears at the beginning of a line.

```
< {a-z} +           matches identifiers only at the beginning of lines
```

- A pattern followed by the operator '>' matches a character sequence only if it appears at the end of a line.

```
" " + >           matches trailing spaces
< C ANY * >       matches FORTRAN comment lines
```

- A pattern followed by the operator '/' and a regular expression matches a character sequence only if it is followed by a character sequence that is matched by the regular expression behind the operator '/'.

```
{0-9} + / ".."    matches numbers, but only if followed by two dots
```

- Several patterns that share a common action can be given in a comma separated list, thus the action has to be specified only once.

```
 ' - {\n'} * ', \" - {\n"} * \"
                  matches both possible forms of Modula-2 strings
```

3.3. Ambiguous Specifications

Rex can handle ambiguous specifications. When more than one expression can match the current input, Rex chooses as follows:

- The longest match is preferred.
- Among rules which match the same number of characters, the rule given first is preferred.

The length of a match is the number of matched characters plus the number of characters matched by the regular expression following the "right context" operator '/' if applicable.

Example:

```
{0-9} + / ".."       : { return SymDecimal; }
{0-9} + "." {0-9} * : { return SymReal    ; }
".."                 : { return SymRange  ; }
"."                  : { return SymDot    ; }
```

Suppose the right context of the first rule above is missing. The input

```
1..
```

would be recognized as SymReal and SymDot because SymReal matches two characters. To get the right solution the right context is necessary. Now the input is recognized as SymDecimal and SymRange because SymDecimal matches 3 characters.

Example:

```
BEGIN   : { return SymBegin; }
END     : { return SymEnd  ; }
{A-Z} + : { return SymIdent; }
```

The rules for keywords should be given before the rule for identifiers. Otherwise the keywords would be recognized as identifiers.

3.4. Definitions

Regular expressions can be given names. This serves to avoid duplication of regular expressions or to increase the expressive power of a specification. After the keyword DEFINE a list of identifiers can be associated with regular expressions. Defined identifiers appearing within regular expressions are replaced by the regular expression given in the definition. Undefined identifiers are treated as strings. The identifier ANY is predefined to match any character except newline.

Example:

```
letter           = { A-Z a-z } .
digit            = { 0-9 }     .
string_character = - { " \n }  .
ANY              = - { \n }    .
```

3.5. Start States

For complex tasks Rex offers a facility called "start states". Usually the generated scanner is always in the standard state called STD and all specified patterns are recognized. In general the scanner is allowed to change its state between an arbitrary number of user defined states. The patterns can be specified to be recognized only in certain states. Initially the scanner is in the standard start state STD. There are special statements to change the state of the scanner (see section 4.4.). They can be used in the actions of the rules.

Start states have to be defined by giving a list of identifiers after the keyword START. The identifiers may be separated by commas. The standard state STD is predefined.

- A pattern without given start states is recognized in every start state the scanner is in.
- A pattern preceded by a list of start states (inclosed in '#' characters) is recognized only if the scanner is in one of the listed start states. Again the listed start states may be separated by commas.
- A pattern preceded by the keyword NOT and a list of start states (enclosed in '#' characters) is recognized only if the scanner is in a start state not listed.

Example:

```
START comment
RULE
              "(*"         : {++ level; yyStart (comment);}
#comment#     "*)"         : {-- level; if (level == 0) yyStart (STD);}
#comment#     "(" | "*" | - {*()} + : {}
#STD#         {0-9} +      : {return SymNumber;}
```

The above example shows how to handle nested comments in a Modula-2 scanner. The rule for opening comment brackets is recognized in all states. The nesting level is increased and we change the start state to *comment* with the predefined statement yyStart. Closing comment brackets are recognized only if the scanner is in start state *comment.* Upon their recognition the nesting level is decreased. Should the nesting level reach zero the comment is finished and we change the state back to STD using yyStart again. While the scanner is in start state *comment* everything except opening and closing comment brackets is skipped by specifying an empty action. The last rule specifying the structure of decimal numbers is recognized only in the start state STD.

The problem of how to declare the variable for counting the nesting level of comments is solved in section 3.7.

3.6. Scanner Name

A specification may be optionally headed by a name for the scanner to be generated:

Example:

```
SCANNER lexer
```

The identifier is used to derive the names of the scanner and source modules, the file name for the scanner tables, and a prefix for the objects exported by the scanner. If the name is missing it defaults to *Scanner*. In the following we refer to this name by <Scanner>. The prefixes

<Scanner> and <Scanner>_ are generated only if this clause is present. Otherwise they are omitted in order to be compatible with former versions of Rex.

3.7. Target Code

The actions associated with regular expressions may need variables or in general arbitrary declarations to perform their task. A scanner specification may be preceded by several kinds of sections written in the target language. The syntax rules for actions apply to these sections, too. These sections are copied unchanged and unchecked to the generated scanner at the following places:

- Target code after the keyword EXPORT is included in the specification part (definition module) of the generated scanner. It allows to extend the set of objects exported by the scanner module. If not given it is predefined as described below.
- Target code after the keyword GLOBAL is included in the scanner module at level 0, that is the extent of variables given in this section is the run time of the whole program. If not given it is predefined as described below.
- Target code after the keyword LOCAL is included in the scanner routine called <Scanner>_GetToken (at level 1), that is the extent of variables given in this section is one invocation of this routine.
- Target code after the keyword BEGIN is included in the routine <Scanner>_BeginScanner which may be called to initialize the data structures declared in the sections EXPORT and GLOBAL.
- Target code after the keyword CLOSE is included in the routine <Scanner>_CloseScanner which may be called after scanning is finished. This statements can be used to finalize the data structures declared in the sections EXPORT and GLOBAL.
- Target code after the keyword DEFAULT is included in the scanner routine to be executed whenever a character is not matched by one of the regular expressions. It can be used to detect illegal characters for example. If not given it is predefined as described below.
- Target code after the keyword EOF is included in the scanner routine to be executed upon reaching the end of the input. It can be used to return a value different from the predefined one (<Scanner>_EofToken = 0) or to check for unclosed comments or strings for example.

If the EXPORT, GLOBAL, and DEFAULT sections are not used the following predefined declarations are included:

If the target language is C:

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

GLOBAL {
void <Scanner>_ErrorAttribute (Token, Attribute)
   int Token;
   <Scanner>_tScanAttribute * Attribute;
   { }
}

DEFAULT {
   yyEcho;
}
```

If the target language is Modula-2:

```
EXPORT {
IMPORT Positions;
TYPE tScanAttribute = RECORD Position: Positions.tPosition; END;
PROCEDURE ErrorAttribute (Token: INTEGER; VAR Attribute: tScanAttribute);
}

GLOBAL {
PROCEDURE ErrorAttribute (Token: INTEGER; VAR Attribute: tScanAttribute);
   BEGIN
   END ErrorAttribute;
}

DEFAULT {
   yyEcho;
}
```

These two sections import the type tPosition from a module named Positions and they declare the type <Scanner>_tScanAttribute as well as the procedure <Scanner>_ErrorAttribute. These items are needed in combination with parser generators. A variable called <Scanner>_Attribute of type <Scanner>_tScanAttribute is used to communicate additional properties of the tokens from the scanner to the parser. The type <Scanner>_tScanAttribute has to be a struct (record) type with at least one member (field) called Position of type tPosition. tPosition has to be a struct (record) type with at least two members (fields) called Line and Column (see section 3.8.). It can be imported from the predefined module Positions or from a user modified version of it.

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 types tPosition and <Scanner>_tScanAttribute are predefined as given above and the procedure <Scanner>_ErrorAttribute is empty. If only one of the sections EXPORT or GLOBAL is used, it has to contain declarations consistent with the remaining predefined ones.

3.8. Source Position

The generated scanners automatically compute the line and column position of every token. This position can be accessed via the fields Position.Line and Position.Column of the global variable <Scanner>_Attribute as described in the section about the Scanner Interface. The source position is computed automatically if the action of a rule is preceded by a colon like in all the examples so far. However, if the character ’-’ is appended to the colon, the calculation of the source position can be disabled for a rule.

There are mainly two reasons for not to compute the position. First, some "compound" tokens have to be recognized by the combination of several rules (usually in connection with a start state). In order to get the correct position, which is the position yielded by the first rule, the calculation of the position has to be disabled for the following rules.

Example (Pascal strings):

```
START string
RULE
#STD#    ’               :  {yyStart (string);}
#string# - {’\t\n} +     :- {}
#string# ’’              :- {}
#string# ’               :- {yyStart (STD); return SymString;}
```

Second, there is no need to calculate the source position in rules that skip input characters without returning a token. In this case disabling the computation of the position yields an increase in run time efficiency. The typical examples are comments. The example given in the chapter about Start States should be rewritten as follows:

Example (Modula-2 comments):

```
START comment
RULE
             "(*"           :- {++ level; yyStart (comment);}
#comment#    "*)"           :- {-- level; if (level == 0) yyStart (STD);}
#comment#    "(" | "*" | - {*(\t\n} + :- {}
```

4. Predefined Items

Rex knows several predefined items described in the next sections.

4.1. Definitions

The identifier ANY is predefined to match one arbitrary character except newline.

```
DEFINE ANY = - { \n } .
```

4.2. Start States

The identifier STD is predefined to denote the standard start state of Rex. The generated scanners are initially in this state.

```
START STD
```

4.3. Rules

The 4 for rules given below are predefined after the user specified rules. By giving own rules the user can overwrite these because of the strategy to solve ambiguities. The predefined rules help to calculate the line and column positions and to skip blanks efficiently.

```
RULE
" "      :- {}
\t       :- {yyTab;}
\n       :- {yyEol (0);}
ANY      :- {yyEcho;}
```

4.4. Action Statements

The following statements can be used within the actions associated with regular expressions:

<Scanner>_GetWord (v);
: This statement gives access to the matched character sequence.
In C the sequence is returned in the variable v which must be of type char v [ ].
Additionally the length of the sequence is returned as result of the function.
In Modula-2 the sequence is returned in the variable v which must be of type Strings.tString.

<Scanner>_GetLower (v);
: Like <Scanner>_GetWord, except that every letter is normalized to lower case.

<Scanner>_GetUpper (v);
: Like <Scanner>_GetWord, except that every letter is normalized to upper case.

yyEcho;
: The matched character sequence is printed on standard output.

yyLess (n);
: The matched character sequence is truncated to the first n characters. The other characters are rescanned for the next character sequence.

yyStart (s);	The start state is changed to state s.
yyPrevious;	The start state is changed to the state valid before the last execution of yyStart or yyPrevious.
yyStartState	This is not a statement but an expression of type short or SHORTCARD, respectively, whose value is the current start state. It can be used to execute different statements in one action depending on the current start state.
yyTab;	This statement should be used if a regular expression is specified by the user to process tab characters. Its purpose is to update the internal variable to calculate the column position of tokens. yyTab works only if the tab character exclusively is specified by a rule.
yyTab1 (a);	Like yyTab this statement should be used if a regular expression is specified by the user to process tab characters. Its purpose is to update the internal variable to calculate the column position of tokens. yyTab1 works if the tab character is embedded in other characters. The parameter a must specify the number of characters before the tab character.
yyEol (n);	This statement should be used if a regular expression is specified by the user to process newline characters. Its purpose is to update the internal variables to calculate the line and column position of tokens. yyEol should be executed once for every newline character matched. The parameter n should specify the number of characters matched after the last newline character. In simple cases where the pattern consists only of a newline character one invocation of yyEol (0); is sufficient.

5. Interface of the Generated Scanners

The scanners generated by Rex offer an interface to be used by a main program like e. g. a parser and they require a source module for blocked input of characters to obey a certain interface. The structure of these two interfaces is independent from a specific target language. As the syntactic details vary from one target language to another we discuss the interfaces in the following target language specific chapters.

5.1. C

It has been already mentioned that the prefixes <Scanner> and <Scanner>_ are generated only if the keyword SCANNER is present. Otherwise they are omitted in order to be compatible with former versions of Rex.

5.1.1. Scanner Interface

The scanners generated by Rex offer an interface given by the following specification file named <Scanner>.h:

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

# define                <Scanner>_EofToken       0

extern  char *          <Scanner>_TokenPtr       ;
extern  short           <Scanner>_TokenLength    ;
extern  <Scanner>_tScanAttribute <Scanner>_Attribute;
extern  void         (* <Scanner>_Exit) ()       ;

extern  void            <Scanner>_BeginScanner    ();
extern  void            <Scanner>_BeginFile       (char * FileName);
extern  int             <Scanner>_GetToken        ();
extern  int             <Scanner>_GetWord         (char * Word);
extern  int             <Scanner>_GetLower        (char * Word);
extern  int             <Scanner>_GetUpper        (char * Word);
extern  void            <Scanner>_CloseFile       ();
extern  void            <Scanner>_CloseScanner    ();
```

- The procedure <Scanner>_GetToken is the central scanning routine. It returns the next token found in the input or whatever is specified in the actions associated with the regular expressions.
- The procedure <Scanner>_BeginFile may be called to open an input file or a nested include file. It has one parameter of type 'char *' (string) which specifies the file name. Include files up to a nesting depth of 15 can be processed. If not called input is read from standard input.
- The procedure <Scanner>_CloseFile may be called to close the actual input file (before reaching end of file). <Scanner>_CloseFile is called automatically by the scanner upon reaching end of file.
- The procedure <Scanner>_BeginScanner may be called to initialize user data.
- The procedure <Scanner>_CloseScanner may be called to finalize user data.
- The procedures <Scanner>_GetWord, <Scanner>_GetLower, and <Scanner>_GetUpper allow access to the matched character sequence as described in section 4.4.
- Alternatively, the matched character sequence can be accessed using the variables <Scanner>_TokenPtr and <Scanner>_TokenLength. <Scanner>_TokenPtr points to the beginning of the matched character sequence. <Scanner>_TokenLength specifies the number of the matched characters. Note, the matched character sequence is not terminated by a '\0' character.
- The variable <Scanner>_Attribute is supposed to communicate additional properties of the last token. The value must be provided by appropriate action statements. This variable is of type <Scanner>_tScanAttribute which has to be a struct type with at least one member called Position of type tPosition. tPosition has to be a struct type with at least two members called Line and Column. The values of Line and Column are computed by the scanner, automatically. They indicate the source position of the actual token. The position of a token is the position of the first character of the token. For exceptions see section 3.8. The types <Scanner>_tScanAttribute and tPosition are predefined as given above. The definitions of these types can be changed as described in section 3.7.
- 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 <Scanner>_Attribute a default value for the additional properties of the to-

ken Token.

- The variable <Scanner>_Exit refers to a procedure which is called upon an internal error in the scanner. The default procedure terminates the program execution. The variable can be changed to achieve a different behaviour.
- If the scanner reaches the end of the input it returns the special token called <Scanner>_EofToken which is encoded by 0.

5.1.2. Source Interface

The scanners generated by Rex need a source module for blocked input of characters. Rex can provide a prototype source module which reads from standard input or any file. It is contained in the files <Scanner>Source.h and <Scanner>Source.c. The specification file <Scanner>Source.h consists of something like:

```
extern int  <Scanner>_BeginSource  (char * FileName);
extern int  <Scanner>_GetLine      (int File, char * Buffer, int Size);
extern void <Scanner>_CloseSource  (int File);
```

- <Scanner>_BeginSource is called from the scanner in order to open files or to initialize any other source of input. If not called input is read from standard input.
- <Scanner>_GetLine is called to fill a buffer starting at address 'Buffer' with a block of maximal 'Size' characters. Lines are terminated by newline characters (ASCII = 0xa). <Scanner>_GetLine returns the number of characters transferred. Reasonable block sizes are between 128 and 2048 or the length of a line. Smaller block sizes - especially block size 1 - will drastically slow down the scanner.
- <Scanner>_CloseSource is called from the scanner at end of file respectively at end of input. It can be used to close files.

The body in the file <Scanner>Source.c has the following contents:

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

int <Scanner>_BeginSource (FileName)
   char * FileName;
{  return OpenInput (FileName); }

int <Scanner>_GetLine (File, Buffer, Size)
   int File; char * Buffer; int Size;
{
   register int n = Read (File, Buffer, Size);
# ifdef Dialog
# define IgnoreChar ' '
/* Add dummy after newline character in order to supply a lookahead for rex. */
/* This way newline tokens are recognized without typing an extra line.      */
   if (n > 0 && Buffer [n - 1] == '0) Buffer [n ++] = IgnoreChar;
# endif
   return n;
}

void <Scanner>_CloseSource (File)
   int File;
{  Close (File); }
```

The newline character may constitute a token of its own in applications such as dialog programs. Like for every other token, Rex needs at least a look-ahead of one character to recognize this token. Therefore the user has to type not only one extra character but a complete extra input line because usually input is line buffered by the operating system. This behaviour is undesirable. The problem can be solved by compiling the file <Scanner>Source.c with the option -DDialog. This variant adds a dummy character after the newline character to serve as lookahead. The dummy character should be a character that is ignored such as e. g. a blank.

5.1.3. Scanner Driver

To test a generated scanner a main program is necessary. Rex can provide a minimal main program in the file <Scanner>Drv.c which can serve as test driver. It counts the tokens and looks like the following:

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

main ()
{
   int Token, Count = 0;
   char Word [256];

   <Scanner>_BeginScanner ();
   do {
      Token = <Scanner>_GetToken ();
      Count ++;
# ifdef Debug
      if (Token != <Scanner>_EofToken) (void) <Scanner>_GetWord (Word);
      else Word [0] = '\0';
      WritePosition (stdout, <Scanner>_Attribute.Position);
      (void) printf ("%5d %s\n", Token, Word);
# endif
   } while (Token != <Scanner>_EofToken);
   <Scanner>_CloseScanner ();
   (void) printf ("%d\n", Count);
   return 0;
}
```

5.2. Modula-2

5.2.1. Scanner Interface

The scanners generated by Rex offer an interface given by the following definition module named <Scanner>.md:

```
DEFINITION MODULE <Scanner>;

IMPORT Positions, Strings;

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

CONST EofToken  = 0;

VAR TokenLength : INTEGER;
VAR Attribute   : tScanAttribute;
VAR ScanTabName : ARRAY [0 .. 127] OF CHAR;
VAR Exit        : PROC;

PROCEDURE BeginScanner  ;
PROCEDURE BeginFile     (FileName: ARRAY OF CHAR);
PROCEDURE GetToken      (): INTEGER;
PROCEDURE GetWord       (VAR Word: Strings.tString);
PROCEDURE GetLower      (VAR Word: Strings.tString);
PROCEDURE GetUpper      (VAR Word: Strings.tString);
PROCEDURE CloseFile     ;
PROCEDURE CloseScanner  ;

END <Scanner>.
```

- The procedure GetToken is the central scanning routine. It returns the next token found in the input or whatever is specified in the actions associated with the regular expressions.
- The array ScanTabName specifies the name of the file containing the scanner tables. It is initialized with the string "Scan.Tab". Therefore, the scanner 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 GetToken the first time.
- The procedure BeginFile may be called to open an input file or a nested include file. The parameter FileName specifies the file name. Include files up to a nesting depth of 15 can be processed. If not called input is read from standard input.
- The procedure CloseFile may be called to close the actual input file (before reaching end of file). CloseFile is called automatically by the scanner upon reaching end of file.
- The procedure BeginScanner may be called to initialize user data.
- The procedure CloseScanner may be called to finalize user data.
- The procedures GetWord, GetLower, and GetUpper allow access to the matched character sequence as described in section 4.4.
- The variable TokenLength specifies the number of the matched characters.
- The variable Attribute is supposed to communicate additional properties of the last token. The value must be provided by appropriate action statements. This variable is of type tScanAttribute which has to be a record type with at least one field called Position of type tPosition. tPosition has to be a record type with at least two fields called Line and Column. The values of Line and Column are computed by the scanner, automatically. They indicate the source position of the actual token. The position of a token is the position of the first character of the token. For exceptions see section 3.8. The types tScanAttribute and tPosition are predefined as given above. The definitions of these types can be changed as described in section 3.7.
- 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 the token Token.

- The variable Exit refers to a procedure which is called upon an internal error in the scanner. The default procedure terminates the program execution. The variable can be changed to achieve a different behaviour.
- If the scanner reaches the end of the input it returns the special token called EofToken which is encoded by 0.

5.2.2. Source Interface

The scanners generated by Rex need a source module for blocked input of characters. Rex can provide a prototype source module which reads from standard input. It is contained in the files <Scanner>Source.md and <Scanner>Source.mi. The definition module in the file <Scanner>Source.md has the following contents:

```
DEFINITION MODULE <Scanner>Source;

FROM SYSTEM     IMPORT ADDRESS;
FROM System     IMPORT tFile;

PROCEDURE BeginSource (FileName: ARRAY OF CHAR): tFile;
PROCEDURE GetLine (File: tFile; Buffer: ADDRESS; Size: CARDINAL): INTEGER;
PROCEDURE CloseSource (File: tFile);

END <Scanner>Source.
```

- BeginSource is called from the scanner in order to open files or to initialize any other source of input. If not called input is read from standard input.
- GetLine is called to fill a buffer starting at address ’Buffer’ with a block of maximal ’Size’ characters. Lines are terminated by newline characters (ASCII = 12C). GetLine returns the number of characters transferred. Reasonable block sizes are between 128 and 2048 or the length of a line. Smaller block sizes - especially block size 1 - will drastically slow down the scanner.
- CloseSource is called from the scanner at end of file respectively at end of input. It can be used to close files.

The implementation module in the file <Scanner>Source.mi has the following contents:

```
IMPLEMENTATION MODULE <Scanner>Source;

FROM SYSTEM     IMPORT ADDRESS;
FROM System     IMPORT tFile, OpenInput, Read, Close;

PROCEDURE BeginSource (FileName: ARRAY OF CHAR): tFile;
   BEGIN
      RETURN OpenInput (FileName);
   END BeginSource;

PROCEDURE GetLine (File: tFile; Buffer: ADDRESS; Size: CARDINAL): INTEGER;
   CONST IgnoreChar = ' ';
   VAR n        : INTEGER;
   VAR BufferPtr: POINTER TO ARRAY [0..30000] OF CHAR;
   BEGIN
   (* # ifdef Dialog
      n := Read (File, Buffer, Size);
(* Add dummy after newline character in order to supply a lookahead for rex. *)
(* This way newline tokens are recognized without typing an extra line.      *)
      BufferPtr := Buffer;
      IF (n > 0) AND (BufferPtr^[n - 1] = 012C) THEN
         BufferPtr^[n] := IgnoreChar; INC (n); END;
      RETURN n;
      # else *)
      RETURN Read (File, Buffer, Size);
   (* # endif *)
   END GetLine;

PROCEDURE CloseSource (File: tFile);
   BEGIN
      Close (File);
   END CloseSource;

END <Scanner>Source.
```

The newline character may constitute a token of its own in applications such as dialog programs. Like for every other token, Rex needs at least a look-ahead of one character to recognize this token. Therefore the user has to type not only one extra character but a complete extra input line because usually input is line buffered by the operating system. This behaviour is undesirable. The problem can be solved by modifying the procedure GetLine in the file <Scanner>Source.mi. The variant in the comment (* # ifdef Dialog ... # else *) adds a dummy character after the newline character to serve as lookahead. The dummy character should be a character that is ignored such as e. g. a blank.

5.2.3. Scanner Driver

To test a generated scanner a main program is necessary. Rex can provide a minimal main program in the file <Scanner>Drv.mi which can serve as test driver. It counts the tokens and looks like the following:

```
MODULE <Scanner>Drv;

FROM <Scanner>  IMPORT BeginScanner, GetToken, GetWord, Attribute, EofToken,
                       CloseScanner;
FROM Strings    IMPORT tString, WriteL;
FROM IO         IMPORT StdOutput, WriteI, WriteC, WriteNl, CloseIO;
FROM Positions  IMPORT WritePosition;
FROM System     IMPORT Exit;

VAR Token       : INTEGER;
    Word        : tString;
    Debug       : BOOLEAN;
    Count       : INTEGER;
```

```
BEGIN
   Debug := FALSE;
   Count := 0;
   BeginScanner;
   REPEAT
      Token := GetToken ();
      INC (Count);
      IF Debug THEN
         GetWord (Word);
         WritePosition (StdOutput, Attribute.Position);
         WriteI (StdOutput, Token, 5);
         WriteC (StdOutput, ' ');
         WriteL (StdOutput, Word);
      END;
   UNTIL Token = EofToken;
   CloseScanner;
   WriteI (StdOutput, Count, 0);
   WriteNl (StdOutput);
   CloseIO;
   Exit (0);
END <Scanner>Drv.
```

6. Usage

NAME

rex – generator of lexical analysers

SYNOPSIS

rex [-options] [-l*dir*] [file]

DESCRIPTION

Rex generates programs to be used in lexical analysis of text. A typical application is the generation of scanners for compilers. The input *file* contains regular expressions to be searched for, and actions written in C or Modula-2 to be executed when strings according to the expressions are found. Unrecognized portions of the input are copied to standard output. To be able to recognize tokens depending on their context, *Rex* provides start states to handle left context and the right context can be specified by an additional regular expression. If several regular expressions match the input characters, the longest match is preferred. If there are still several possibilities, the regular expression given first in the specification is chosen.

Rex generated scanners automatically provide the line and column position of each token. For languages like Pascal and Ada where the case of letters is insignificant tokens can be normalized to lower or upper case. There are predefined rules to skip white space like blanks, tabs, or newlines and there is a mechanism to handle include files. The generated scanners are table-driven deterministic finite automatons.

OPTIONS

a generate all (= sdm)

m generate a lexical analyser in Modula-2 (default)

c generate a lexical analyser in C

d generate a definition module for the lexical analyser

s generate support modules:
- a source module for input
- a main program to be used as test driver

r reduce the number of generated case/switch labels. Might be necessary due to compiler restrictions. Effects: slower scanner (2-4%), larger tables, same scanner size.

i use ISO 8 bit code (default: ASCII 7 bit code)

o optimize table size
Effects: slower scanner (0-15%), small tables, long generation time (factor 1-10)

n do not optimize table size
Effects: fast scanner, large tables (factor 1-10), short generation time

default: improve table size
Effects: slower scanner (0-5%), medium size tables (factor 1-2), medium generation time (factor 1-2)

w suppress warnings

g generate # line directives

b do not partition charcater set into blocks

1 print statistics about the generated lexical analyser

ldir *dir* is the directory where Rex finds its table and data files

FILES

if output is in C:

<Scanner>.h	specification of the generated scanner
<Scanner>.c	body of the generated scanner
<Scanner>Source.h	specification of support module source
<Scanner>Source.c	body of support module source
<Scanner>Drv.c	main program to serve as test driver

if output is in Modula-2:

<Scanner>.md	definition module of the generated scanner
<Scanner>.mi	implementation module of the generated scanner
<Scanner>Source.md	definition module of support module source
<Scanner>Source.mi	implementation module of support module source
<Scanner>Drv.mi	main program to serve as test driver
<Scanner>.Tab	tables to control the generated scanner

SEE ALSO

J. Grosch: "Rex - A Scanner Generator", GMD Forschungsstelle an der Universitaet Karlsruhe, Compiler Generation Report No. 5, 1987

J. Grosch: "Efficient Generation of Lexical Analysers", Software - Practice & Experience, 19 (11), 1089-1103, Nov. 1989

7. Implementation

Rex is implemented by a 5,000 line Modula-2 program. The program makes heavy use of a library of reusable Modula-2 modules currently comprising 3,000 lines of code [Gro87]. Of the 5,000 lines of Rex 1,500 lines are generated by tools:

- 500 lines for the scanner are generated by Rex itself.
- 1000 lines for the parser are generated by the LALR(1) parser generator *lalr*.

How can Rex generate a part of itself before its existence? Well, the scanner has been bootstrapped using LEX. The first version of the scanner was a separate C program generated by LEX which wrote the internal representation of the tokens on a file. A simple hand written scanner read the tokens from this file during construction of Rex. After Rex was operational it could generate its own scanner in Modula-2.

And how is Rex working? It differentiates between constant regular expressions and non-constant ones as defined in [Gro89]. The non-constant regular expressions constitute a nondeterministic finite automaton. The so-called subset construction algorithm is used for conversion into a deterministic finite automaton. Then an algorithm to minimize the number of states is applied. After extending the automaton to a tunnel automaton the constant regular expressions are added in linear time using the algorithm described in [Gro89]. The sparse matrix to control the automaton is compressed into a data structure called "comb vector" [ASU86] to save space.

The key to the performance of scanners generated by Rex lies in the following facts:

- access to the "comb vector" table is fast
- input happens rarely because blocks of characters are transferred
- no check for the last character of a block is necessary because of the sentinel technique used
- the same holds for the check of stack underflow for the stack to record the passed states
- the treatment of right context is efficient and only necessary in a few cases because partial evaluation has been applied

8. Differences to LEX

Some specialists might want to know about the differences between Rex and LEX [Les75] (see Table):

Table: Syntactical differences between Rex and LEX:

Meaning	LEX	Rex
delimiter for character classes	[ ]	{ }
complement of character classes	[ˆ]	- { }
any character	.	ANY
left justification	ˆ	<
right justification	$	>
replicator	{n}	[n]
replicator	{m,n}	[m-n]
delimiter for start states	< >	# #
escape representation for characters	\octal	\decimal
scanner routine	yylex	<Scanner>_GetToken
access to matched string	yytext	<Scanner>_GetWord ( )
length of matched string	yyleng	result of <Scanner>_GetWord ( )
output of matched string	ECHO	yyEcho
retain part of matched string	yyless	yyLess
initial start state	INITIAL	STD
change of start state	BEGIN	yyStart ( )

Advantages of Rex:

+ The standard or initial start state has a documented name: STD.
+ The list of start states can be inverted using the operator NOT to specify that a rule is valid in all states except the listed ones.
+ The specifications can be written unformatted - white space in the form of blanks, tabs, and newlines is skipped.
+ Identifiers used to refer to named regular expressions are written without enclosing braces '{' '}'.
+ Rex automatically calculates the source position of the tokens in the fields Line and Column of the variable <Scanner>_Attribute.
+ There are predefined rules to skip the white space characters.
+ Include files up to a nesting depth of 15 can be processed.
+ Routines are provided to normalize tokens to upper or lower case characters.
+ No adjustment of the internal data structures are necessary to be able to process large specifications.

Disadvantages of Rex:

– The action statement yymore is not available.
– The action statement REJECT is not available - Rex can only find one solution and not all like LEX.
– The redirection of input with the procedure yywrap is not available.
– The character set is fixed to the one of the host computer. There is no way to specify a different character set to be able to generate scanners for target computers with a different character set.

Appendix 1: Syntax of the Specification Language

```
specification    : [name] [code] [define] [start] rules
                 .
name             : SCANNER [Ident]
                 .
code             :
                 | code EXPORT  TargetCode
                 | code GLOBAL  TargetCode
                 | code LOCAL   TargetCode
                 | code BEGIN   TargetCode
                 | code CLOSE   TargetCode
                 | code DEFAULT TargetCode
                 | code EOF     TargetCode
                 .
define           : DEFINE definition *
                 .
start            : START identList
                 .
rules            : RULE  rule *
                 | RULES rule *
                 .
definition       : Ident '=' regExpr '.'
                 .
identList        : Ident
                 | identList Ident
                 | identList ',' Ident
                 .
rule             : patternList ':'  TargetCode
                 | patternList ':-' TargetCode
                 .
patternList      : pattern
                 | patternList ',' pattern
                 .
pattern          : [startStates] ['<'] regExpr ['/' regExpr] ['>']
                 .
startStates      : '#' identList '#'
                 | NOT '#' identList '#'
                 .
regExpr          : regExpr '|' regExpr
                 | regExpr regExpr
                 | regExpr '+'
                 | regExpr '*'
                 | regExpr '?'
                 | regExpr '[' Number ']'
                 | regExpr '[' Number '-' Number ']'
                 | '(' regExpr ')'
                 | charSet
                 | Char
                 | Ident
                 | String
                 | Number
                 .
charSet          : '-' charSet
                 | '{' range * '}'
                 .
range            : Char
                 | Char '-' Char
                 .
Char             : character
```

```
                 | '\' digit +
                 | '\' n
                 | '\' t
                 | '\' v
                 | '\' b
                 | '\' r
                 | '\' f
                 | '\' character
                 .
Ident            : letter letter_or_digit *
                 .
letter_or_digit : letter
                 | digit
                 | '_'
                 .
String           : '"' character * '"'
                 .
Number           : digit +
                 .
Target_code      : '{' character * '}'
                 .
```

Appendix 2: Example Specification of a Modula-2-Scanner in C

```
GLOBAL  {
# include "Memory.h"
# include "StringMem.h"
# include "Idents.h"

int level = 0;

void ErrorAttribute (Token, Attribute)
   int Token;
   tScanAttribute Attribute;
   {
   }
}

LOCAL   {
   char          Word [256];
   tIdent        ident   ;
   tStringRef    ref     ;
   int           length  ;
}

DEFAULT {
   printf ("illegal character: "); yyEcho; printf ("\n");
}

DEFINE

   digit         = {0-9}          .
   letter        = {a-z A-Z}      .
   cmt           = - {*(\t\n}     .

START   comment

RULE
           "(*"          :- {++ level; yyStart (comment);}
#comment#  "*)"          :- {-- level; if (level == 0) yyStart (STD);}
#comment#  "(" | "*" | cmt + :- {}

   /* The procedure PutString is imported from the module StringMem(ory).
      It is used to store the string representation of some tokens.       */

#STD# digit +           ,
#STD# digit + / ".."    : {length = GetWord (Word);
                           ref = PutString (Word, length);
                           return 1;}
#STD# {0-7} + B         : {length = GetWord (Word);
                           ref = PutString (Word, length);
                           return 2;}
#STD# {0-7} + C         : {length = GetWord (Word);
                           ref = PutString (Word, length);
                           return 3;}
#STD# digit {0-9 A-F} * H : {
                           length = GetWord (Word);
                           ref = PutString (Word, length);
                           return 4;}
#STD# digit + "." digit * (E {+\-} ? digit +) ? : {
                           length = GetWord (Word);
                           ref = PutString (Word, length);
                           return 5;}

#STD# ' - {\n'} * '     |
      \" - {\n"} * \"   : {length = GetWord (Word);
                           ref = PutString (Word, length);
                           return 6;}
```

```
#STD# "#"               : {return 7;}
#STD# "&"               : {return 8;}
#STD# "("               : {return 9;}
#STD# ")"               : {return 10;}
#STD# "*"               : {return 11;}
#STD# "+"               : {return 12;}
#STD# ","               : {return 13;}
#STD# "-"               : {return 14;}
#STD# "."               : {return 15;}
#STD# ".."              : {return 16;}
#STD# "/"               : {return 17;}
#STD# ":"               : {return 18;}
#STD# ":="              : {return 19;}
#STD# ";"               : {return 20;}
#STD# "<"               : {return 21;}
#STD# "<="              : {return 22;}
#STD# "<>"              : {return 23;}
#STD# "="               : {return 24;}
#STD# ">"               : {return 25;}
#STD# ">="              : {return 26;}
#STD# "["               : {return 27;}
#STD# "]"               : {return 28;}
#STD# "^"               : {return 29;}
#STD# "{"               : {return 30;}
#STD# "|"               : {return 31;}
#STD# "}"               : {return 32;}
#STD# "~"               : {return 33;}

#STD# AND               : {return 34;}
#STD# ARRAY             : {return 35;}
#STD# BEGIN             : {return 36;}
#STD# BY                : {return 37;}
#STD# CASE              : {return 38;}
#STD# CONST             : {return 39;}
#STD# DEFINITION        : {return 40;}
#STD# DIV               : {return 41;}
#STD# DO                : {return 42;}
#STD# ELSE              : {return 43;}
#STD# ELSIF             : {return 44;}
#STD# END               : {return 45;}
#STD# EXIT              : {return 46;}
#STD# EXPORT            : {return 47;}
#STD# FOR               : {return 48;}
#STD# FROM              : {return 49;}
#STD# IF                : {return 50;}
#STD# IMPLEMENTATION    : {return 51;}
#STD# IMPORT            : {return 52;}
#STD# IN                : {return 53;}
#STD# LOOP              : {return 54;}
#STD# MOD               : {return 55;}
#STD# MODULE            : {return 56;}
#STD# \NOT              : {return 57;}
#STD# OF                : {return 58;}
#STD# OR                : {return 59;}
#STD# POINTER           : {return 60;}
#STD# PROCEDURE         : {return 61;}
#STD# QUALIFIED         : {return 62;}
#STD# RECORD            : {return 63;}
#STD# REPEAT            : {return 64;}
#STD# RETURN            : {return 65;}
#STD# SET               : {return 66;}
```

```
#STD# THEN              : {return 67;}
#STD# TO                : {return 68;}
#STD# TYPE              : {return 69;}
#STD# UNTIL             : {return 70;}
#STD# VAR               : {return 71;}
#STD# WHILE             : {return 72;}
#STD# WITH              : {return 73;}

#STD# letter (letter | digit) * : {
                            ident = MakeIdent (TokenPtr, TokenLength);
                            return 74;}
```

Appendix 3: Example Specification of a Modula-2-Scanner in Modula-2

```
GLOBAL  {
   FROM Strings          IMPORT tString          ;
   FROM StringMem        IMPORT tStringRef       , PutString     ;
   FROM Idents           IMPORT tIdent           , MakeIdent     ;

   VAR level             : CARDINAL;

   PROCEDURE ErrorAttribute (Token: INTEGER; VAR Attribute: tScanAttribute);
      BEGIN
      END ErrorAttribute;
}

LOCAL   {
   VAR
      Word               : tString;
      ident              : tIdent;
      ref                : tStringRef;
}

BEGIN   { level := 0; }

DEFAULT {
IO.WriteS (IO.StdOutput, "illegal character: "); yyEcho; IO.WriteNl (IO.StdOutput);
}

DEFINE

   digit        = {0-9}          .
   letter       = {a-z A-Z}      .
   cmt          = - {*(\t\n}     .

START   comment

RULE
           "(*"          :- {INC (level); yyStart (comment);}
#comment#  "*)"          :- {DEC (level); IF level = 0 THEN yyStart (STD); END;}
#comment#  "(" | "*" | cmt + :- {}

#STD# digit +            ,
#STD# digit + / ".."     : {GetWord (Word);
                            ref := PutString (Word);
                            RETURN 1;}
#STD# {0-7} + B          : {GetWord (Word);
                            ref := PutString (Word);
                            RETURN 2;}
#STD# {0-7} + C          : {GetWord (Word);
                            ref := PutString (Word);
                            RETURN 3;}
#STD# digit {0-9 A-F} * H : {
                            GetWord (Word);
                            ref := PutString (Word);
                            RETURN 4;}
#STD# digit + "." digit * (E {+\-} ? digit +) ? : {
                            GetWord (Word);
                            ref := PutString (Word);
                            RETURN 5;}

#STD# ' - {\n'} * '      |
      \" - {\n"} * \"    : {GetWord (Word);
                            ref := PutString (Word);
                            RETURN 6;}

#STD# "#"                : {RETURN 7;}
#STD# "&"                : {RETURN 8;}
```

```
#STD# "("                : {RETURN 9;}
#STD# ")"                : {RETURN 10;}
#STD# "*"                : {RETURN 11;}
#STD# "+"                : {RETURN 12;}
#STD# ","                : {RETURN 13;}
#STD# "-"                : {RETURN 14;}
#STD# "."                : {RETURN 15;}
#STD# ".."               : {RETURN 16;}
#STD# "/"                : {RETURN 17;}
#STD# ":"                : {RETURN 18;}
#STD# ":="               : {RETURN 19;}
#STD# ";"                : {RETURN 20;}
#STD# "<"                : {RETURN 21;}
#STD# "<="               : {RETURN 22;}
#STD# "<>"               : {RETURN 23;}
#STD# "="                : {RETURN 24;}
#STD# ">"                : {RETURN 25;}
#STD# ">="               : {RETURN 26;}
#STD# "["                : {RETURN 27;}
#STD# "]"                : {RETURN 28;}
#STD# "^"                : {RETURN 29;}
#STD# "{"                : {RETURN 30;}
#STD# "|"                : {RETURN 31;}
#STD# "}"                : {RETURN 32;}
#STD# "~"                : {RETURN 33;}

#STD# AND                : {RETURN 34;}
#STD# ARRAY              : {RETURN 35;}
#STD# BEGIN              : {RETURN 36;}
#STD# BY                 : {RETURN 37;}
#STD# CASE               : {RETURN 38;}
#STD# CONST              : {RETURN 39;}
#STD# DEFINITION         : {RETURN 40;}
#STD# DIV                : {RETURN 41;}
#STD# DO                 : {RETURN 42;}
#STD# ELSE               : {RETURN 43;}
#STD# ELSIF              : {RETURN 44;}
#STD# END                : {RETURN 45;}
#STD# EXIT               : {RETURN 46;}
#STD# EXPORT             : {RETURN 47;}
#STD# FOR                : {RETURN 48;}
#STD# FROM               : {RETURN 49;}
#STD# IF                 : {RETURN 50;}
#STD# IMPLEMENTATION     : {RETURN 51;}
#STD# IMPORT             : {RETURN 52;}
#STD# IN                 : {RETURN 53;}
#STD# LOOP               : {RETURN 54;}
#STD# MOD                : {RETURN 55;}
#STD# MODULE             : {RETURN 56;}
#STD# \NOT               : {RETURN 57;}
#STD# OF                 : {RETURN 58;}
#STD# OR                 : {RETURN 59;}
#STD# POINTER            : {RETURN 60;}
#STD# PROCEDURE          : {RETURN 61;}
#STD# QUALIFIED          : {RETURN 62;}
#STD# RECORD             : {RETURN 63;}
#STD# REPEAT             : {RETURN 64;}
#STD# RETURN             : {RETURN 65;}
#STD# SET                : {RETURN 66;}
#STD# THEN               : {RETURN 67;}
#STD# TO                 : {RETURN 68;}
```

```
#STD# TYPE              : {RETURN 69;}
#STD# UNTIL             : {RETURN 70;}
#STD# VAR               : {RETURN 71;}
#STD# WHILE             : {RETURN 72;}
#STD# WITH              : {RETURN 73;}

#STD# letter (letter | digit) * : {
                            GetWord (Word);
                            ident := MakeIdent (Word);
                            RETURN 74;}
```

Appendix 4: Example Specification of a Scanner for Rex

```
EXPORT  {

FROM Idents      IMPORT tIdent   ;
FROM StringMem   IMPORT tStringRef;
FROM Texts       IMPORT tText    ;
FROM Positions   IMPORT tPosition;

TYPE
   tScanAttribute        = RECORD
              Position   : tPosition     ;
         CASE : INTEGER OF
         | 1: Ident      : tIdent        ;
         | 2: Number     : SHORTCARD     ;
         | 3: String     : tStringRef    ;
         | 4: Ch         : CHAR          ;
         | 5: Text       : tText         ;
         END;
      END;

PROCEDURE ErrorAttribute (Token: INTEGER; VAR Attribute: tScanAttribute);
}

GLOBAL  {

FROM SYSTEM      IMPORT ADDRESS;
FROM Strings     IMPORT tString, Concatenate, Char, SubString,
                        StringToInt, AssignEmpty, Length;
FROM Texts       IMPORT MakeText, Append;
FROM StringMem   IMPORT tStringRef, PutString;
FROM Idents      IMPORT tIdent, MakeIdent, NoIdent;
FROM Errors      IMPORT ErrorMessage, Error;
FROM ScanGen     IMPORT Language, tLanguage;
FROM Positions   IMPORT tPosition;

CONST
   SymIdent              = 1     ;
   SymNumber             = 2     ;
   SymString             = 3     ;
   SymChar               = 4     ;
   SymTargetcode         = 5     ;
   SymScanner            = 37    ;
   SymExport             = 32    ;
   SymGlobal             = 6     ;
   SymLocal              = 31    ;
   SymBegin              = 7     ;
   SymClose              = 8     ;
   SymEof                = 34    ;
   SymDefault            = 36    ;
   SymDefine             = 9     ;
   SymStart              = 10    ;
   SymRules              = 11    ;
   SymNot                = 30    ;
   SymDot                = 12    ;
   SymComma              = 13    ;
   SymEqual              = 14    ;
   SymColon              = 15    ;
   SymColonMinus         = 35    ;
   SymNrSign             = 33    ;
```

```
   SymSlash             = 16    ;
   SymBar               = 17    ;
   SymPlus              = 18    ;
   SymMinus             = 19    ;
   SymAsterisk          = 20    ;
   SymQuestion          = 21    ;
   SymLParen            = 22    ;
   SymRParen            = 23    ;
   SymLBracket          = 24    ;
   SymRBracket          = 25    ;
   SymLBrace            = 26    ;
   SymRBrace            = 27    ;
   SymLess              = 28    ;
   SymGreater           = 29    ;

   BraceMissing         = 13    ;
   UnclosedComment      = 14    ;
   UnclosedString       = 16    ;

VAR
   level        : INTEGER       ;
   string       : tString       ;
   NoString     : tStringRef    ;
   Position     : tPosition     ;

PROCEDURE ErrorAttribute (Token: INTEGER; VAR Attribute: tScanAttribute);
   BEGIN
      CASE Token OF
      | SymIdent        : Attribute.Ident  := NoIdent;
      | SymNumber       : Attribute.Number := 0;
      | SymString       : Attribute.String := NoString;
      | SymChar         : Attribute.Ch     := '?';
      | SymTargetcode   : MakeText (Attribute.Text);
      ELSE
      END;
   END ErrorAttribute;
}

LOCAL   { VAR TargetCode, String, Word: tString; PrevState: SHORTCARD; }

BEGIN   {
   level := 0;
   AssignEmpty (string);
   NoString := PutString (string);
}

EOF     {
   CASE yyStartState OF
   | targetcode ,
     set        : ErrorMessage (BraceMissing    , Error, Attribute.Position);
   | comment    : ErrorMessage (UnclosedComment , Error, Attribute.Position);
   | CStr1, CStr2,
     Str1, Str2 : ErrorMessage (UnclosedString  , Error, Attribute.Position);
   ELSE
   END;
   yyStart (STD);
}

DEFINE
   letter       = {A-Z a-z}     .
   digit        = {0-9}         .
```

```
   string       = - {"\n}        .
   cmtch        = - {*\t\n}      .
   code         = - {{\}\t\n\\'"} .
   StrCh1       = - {'\t\n}      .
   StrCh2       = - {"\t\n}      .
   CStrCh1      = - {'\t\n\\}    .
   CStrCh2      = - {"\t\n\\}    .

START targetcode, set, rules, comment, Str1, Str2, CStr1, CStr2

RULES

#targetcode#    "{"     : {
                           IF level = 0 THEN
                              MakeText (Attribute.Text);
                              AssignEmpty (TargetCode);
                              Position := Attribute.Position;
                           ELSE
                              GetWord (Word);
                              Concatenate (TargetCode, Word);
                           END;
                           INC (level);
                        }

#targetcode#    "}"     :- {
                           DEC (level);
                           IF level = 0 THEN
                              yyStart (PrevState);
                              Append (Attribute.Text, TargetCode);
                              Attribute.Position := Position;
                              RETURN SymTargetcode;
                           ELSE
                              GetWord (Word);
                              Concatenate (TargetCode, Word);
                           END;
                        }

#targetcode#    code +  :- {
                           IF level > 0 THEN
                              GetWord (Word);
                              Concatenate (TargetCode, Word);
                           END;
                        }

#targetcode#    \t      :- {
                           IF level > 0 THEN
                              Strings.Append (TargetCode, 11C);
                           END;
                           yyTab;
                        }

#targetcode#    \n      :- {
                           IF level > 0 THEN
                              Append (Attribute.Text, TargetCode);
                              AssignEmpty (TargetCode);
                           END;
                           yyEol (0);
                        }

#targetcode#    \\ ANY  :- {
                           IF level > 0 THEN
```

```
                                 GetWord (Word);
                                 Strings.Append (TargetCode, Char (Word, 2));
                              END;
                           }

#targetcode#    \\         :- {
                              IF level > 0 THEN
                                 Strings.Append (TargetCode, '\');
                              END;
                           }

#targetcode#    '          : {
                              GetWord (String);
                              IF Language = C
                              THEN yyStart (CStr1);
                              ELSE yyStart (Str1);
                              END;
                           }

#targetcode#    \"         : {
                              GetWord (String);
                              IF Language = C
                              THEN yyStart (CStr2);
                              ELSE yyStart (Str2);
                              END;
                           }

#Str1#  StrCh1 +           ,
#Str2#  StrCh2 +           ,
#CStr1# CStrCh1 + | \\ ANY ? ,
#CStr2# CStrCh2 + | \\ ANY ? :- {GetWord (Word); Concatenate (String, Word);}

#CStr1# \\ \n              ,
#CStr2# \\ \n              :- {GetWord (Word); Concatenate (String, Word); yyEol (0);}

#Str1, CStr1# '            ,
#Str2, CStr2# \"           :- {Strings.Append (String, Char (String, 1));
                               yyPrevious; Concatenate (TargetCode, String);
                           }

#Str1, Str2, CStr1, CStr2# \t :- {Strings.Append (String, 11C); yyTab;}

#Str1, Str2, CStr1, CStr2# \n :- {
                               ErrorMessage (UnclosedString, Error, Attribute.Position);
                               Strings.Append (String, Char (String, 1));
                               yyEol (0); yyPrevious; Concatenate (TargetCode, String);
                           }

#STD, rules#    "/*"       :- {yyStart (comment)    ;}
#comment#       "*" | cmtch +   :- {}
#comment#       "*/"       :- {yyPrevious           ;}

#STD#           EXPORT     : {PrevState := STD; yyStart (targetcode);
                              RETURN SymExport      ;}
#STD#           GLOBAL     : {PrevState := STD; yyStart (targetcode);
                              RETURN SymGlobal      ;}
#STD#           LOCAL      : {PrevState := STD; yyStart (targetcode);
                              RETURN SymLocal       ;}
#STD#           BEGIN      : {PrevState := STD; yyStart (targetcode);
                              RETURN SymBegin       ;}
#STD#           CLOSE      : {PrevState := STD; yyStart (targetcode);
```

```
                          RETURN SymClose       ;}
#STD#           DEFAULT : {PrevState := STD; yyStart (targetcode);
                          RETURN SymDefault     ;}
#STD#           EOF     : {PrevState := STD; yyStart (targetcode);
                          RETURN SymEof         ;}
#STD#           SCANNER : {RETURN SymScanner    ;}
#STD#           DEFINE  : {RETURN SymDefine     ;}
#STD#           START   : {RETURN SymStart      ;}
#STD#           RULE S ?: {yyStart (rules);     RETURN SymRules        ;}
#rules#         \NOT    : {RETURN SymNot        ;}

#STD, rules#    letter (letter | digit | _) * : {
                          GetWord (Word);
                          Attribute.Ident  := MakeIdent (Word);
                          RETURN SymIdent;
                        }

#STD, rules#    digit + : {
                          GetWord (Word);
                          Attribute.Number := StringToInt (Word);
                          RETURN SymNumber;
                        }

#STD, rules#    \" string * \" : {
                          GetWord (Word);
                          SubString (Word, 2, Length (Word) - 1, TargetCode);
                          Attribute.String := PutString (TargetCode);
                          RETURN SymString;
                        }

#STD#           "."     : {RETURN SymDot        ;}
#STD#           "="     : {RETURN SymEqual      ;}
#STD, set#      "}"     : {yyPrevious;          RETURN SymRBrace       ;}
#STD, set, rules# "-"   : {RETURN SymMinus      ;}
#STD, rules#    ","     : {RETURN SymComma      ;}
#STD, rules#    "|"     : {RETURN SymBar        ;}
#STD, rules#    "+"     : {RETURN SymPlus       ;}
#STD, rules#    "*"     : {RETURN SymAsterisk   ;}
#STD, rules#    "?"     : {RETURN SymQuestion   ;}
#STD, rules#    "("     : {RETURN SymLParen     ;}
#STD, rules#    ")"     : {RETURN SymRParen     ;}
#STD, rules#    "["     : {RETURN SymLBracket   ;}
#STD, rules#    "]"     : {RETURN SymRBracket   ;}
#STD, rules#    "{"     : {yyStart (set);       RETURN SymLBrace       ;}
#rules#         "#"     : {RETURN SymNrSign     ;}
#rules#         "/"     : {RETURN SymSlash      ;}
#rules#         "<"     : {RETURN SymLess       ;}
#rules#         ">"     : {RETURN SymGreater    ;}
#rules#         ":"     : {PrevState := rules; yyStart (targetcode);
                          RETURN SymColon       ;}
#rules#         ":-"    : {PrevState := rules; yyStart (targetcode);
                          RETURN SymColonMinus ;}

#STD, set, rules# \\ n  : {Attribute.Ch := 012C; RETURN SymChar;}
#STD, set, rules# \\ t  : {Attribute.Ch := 011C; RETURN SymChar;}
#STD, set, rules# \\ v  : {Attribute.Ch := 013C; RETURN SymChar;}
#STD, set, rules# \\ b  : {Attribute.Ch := 010C; RETURN SymChar;}
#STD, set, rules# \\ r  : {Attribute.Ch := 015C; RETURN SymChar;}
#STD, set, rules# \\ f  : {Attribute.Ch := 014C; RETURN SymChar;}

#STD, set, rules# \\ digit + : {
```

```
                          GetWord (Word);
                          SubString (Word, 2, Length (Word), TargetCode);
                          Attribute.Ch := CHR (CARDINAL (StringToInt (TargetCode)));
                          RETURN SymChar;
                       }

#STD, set, rules# \\ ANY : {
                          GetWord (Word);
                          Attribute.Ch := Char (Word, 2);
                          RETURN SymChar;
                       }

#STD, set, rules# - {\t\n\ \f\r} : {
                          GetWord (Word);
                          Attribute.Ch := Char (Word, 1);
                          RETURN SymChar;
                       }

\f                     :- {}
\r                     :- {}
```

References

[ASU86] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, Reading, MA, 1986.

[Gro87] J. Grosch, Reusable Software - A Collection of Modula-Modules, Compiler Generation Report No. 4, GMD Forschungsstelle an der Universität Karlsruhe, Sep. 1987.

[Gro89] J. Grosch, Efficient Generation of Lexical Analysers, *Software—Practice & Experience 19*, 11 (Nov. 1989), 1089-1103.

[Les75] M. E. Lesk, LEX — A Lexical Analyzer Generator, Computing Science Technical Report 39, Bell Telephone Laboratories, Murray Hill, NJ, 1975.

Contents

1.	Introduction	1
2.	Overview	1
3.	Specification Language	2
3.1.	Lexical Conventions	2
3.2.	Regular Expressions	4
3.3.	Ambiguous Specifications	6
3.4.	Definitions	6
3.5.	Start States	7
3.6.	Scanner Name	7
3.7.	Target Code	8
3.8.	Source Position	9
4.	Predefined Items	10
4.1.	Definitions	10
4.2.	Start States	10
4.3.	Rules	10
4.4.	Action Statements	10
5.	Interface of the Generated Scanners	11
5.1.	C	11
5.1.1.	Scanner Interface	11
5.1.2.	Source Interface	13
5.1.3.	Scanner Driver	14
5.2.	Modula-2	14
5.2.1.	Scanner Interface	14
5.2.2.	Source Interface	16
5.2.3.	Scanner Driver	17
6.	Usage	18
7.	Implementation	19
8.	Differences to LEX	20
	Appendix 1: Syntax of the Specification Language	22
	Appendix 2: Example Specification of a Modula-2-Scanner in C	24
	Appendix 3: Example Specification of a Modula-2-Scanner in Modula-2	27
	Appendix 4: Example Specification of a Scanner for Rex	30
	References	35