Rex
A Scanner Generator

J. Grosch

DR. JOSEF GROSCH

COCOLAB - DATENVERARBEITUNG

GERMANY

Cocktail

Toolbox for Compiler Construction

Rex - A Scanner Generator

Josef Grosch

Aug. 01, 2006

Document No. 5

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

Rex generates program code to be used in lexical analysis of text. A typical application is the generation of scanners for compilers. The generated scanners can handle single byte input as well as Unicode input. 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 one of the target languages (C, C++, Modula-2, Ada, Eiffel or Java) to be executed when regular expressions are matching. 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 1.5 million lines per minute on a SPARC station ELC. (including input from file). If, additionally, hashing of identifiers is performed the speed is around 1.25 million 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, C++, Modula-2, Ada, Eiffel or Java. Rex itself is implemented originally 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. 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. 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

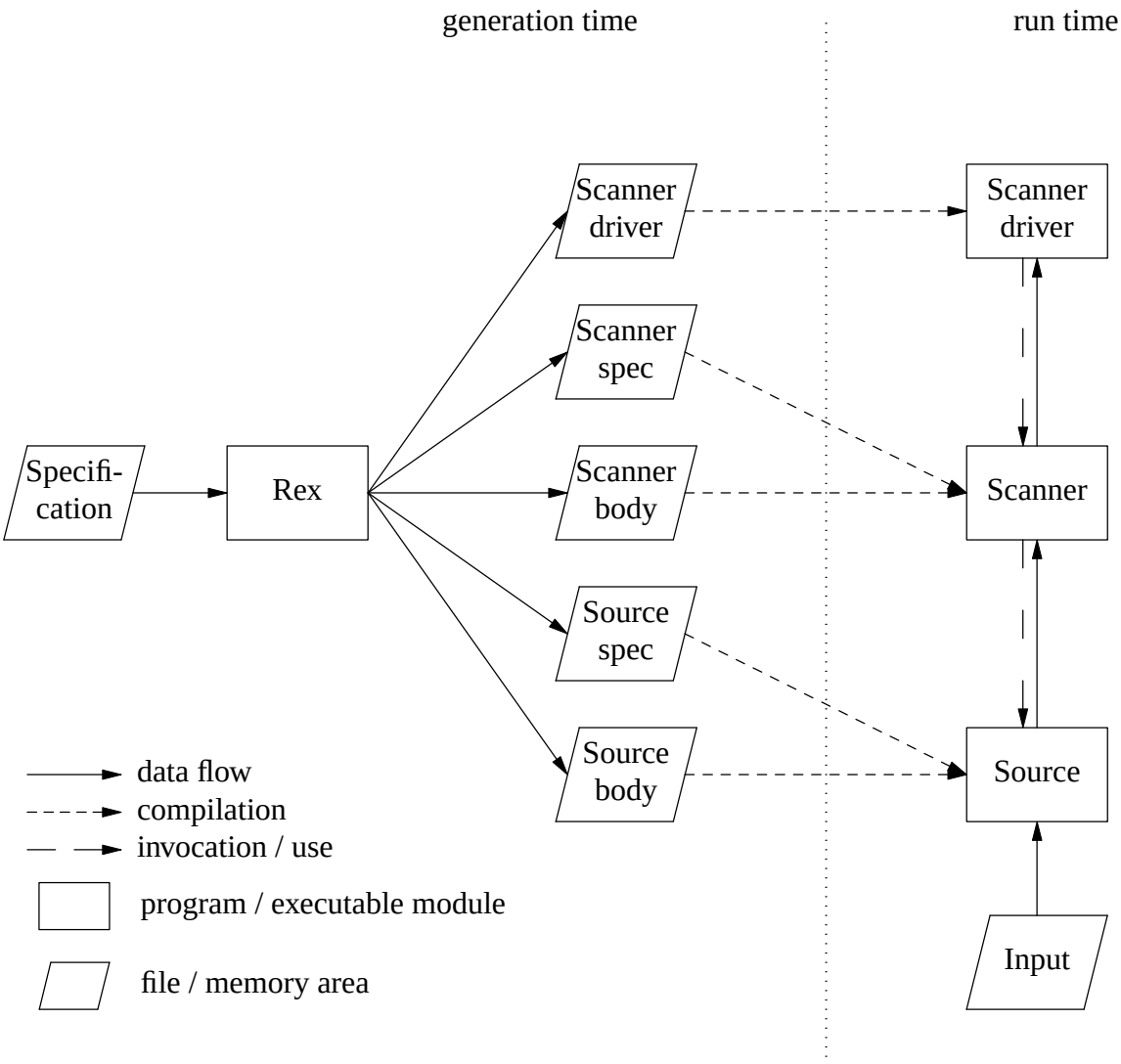

Fig. 1: Rex Overview

replaced by any other main program or subroutine like e. g. a parser.

3. Specification Language

The input of Rex consists mainly of three 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 styles of C or C++: Text included in '/*' and '*/' or from '//' to the end of line is ignored. Comments may not be nested. The specification uses a few keywords which should be escaped if needed as identifiers (see below):

```
BEGIN          CHARACTER_SET 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 available:

bell	BEL	\a
backspace	BS	\b
character tabulation	HT	\t
line feed	LF	\n
line tabulation	VT	\v
form feed	FF	\f
carriage return	CR	\r

Other unprintable characters are represented by the escape character followed either by an integer decimal number or by a hexadecimal number giving the internal encoding. These escape sequences can be used to denote Unicode characters whose representation can take up to 4 bytes.

```
;  \+  \\  \n  \10  \0XAC  \0xabcd  \uabcdef01
```

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 tBegin; }
{ 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 '\'
\0xabcd             matches a Unicode character
\uabcdef01          matches a Unicode 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
{ \0xabcd-\uabcdef01 }matches a set of Unicode characters
- { }               matches all characters
- { \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 tDecimal; }
{0-9} + / ".."      : { return tDecimal; }
{0-9} + "." {0-9} * : { return tReal   ; }
".."                : { return tRange  ; }
"."                 : { return tDot    ; }
```

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

```
1..
```

would be recognized as tReal and tDot because tReal matches two characters. To get the right solution the right context is necessary. Now the input is recognized as tDecimal and tRange because the second rule for tDecimal matches 3 characters.

Example:

```
BEGIN   : { return tBegin; }
END     : { return tEnd  ; }
{A-Z} + : { return tIdent; }
```

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

An analysis that checks a scanner specification for ambiguous rules can be requested with option -p. The result of this analysis is a list of pairs of patterns that are ambiguous with respect to each other.

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. Identifiers have to be declared before use. Undefined identifiers are treated as strings, by default and reported as errors

when option -x is set. 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.

Two kinds of start states are distinguished: "inclusive" start states and "exclusive" start states. This distinction is relevant for patterns given without start states.

Start states have to be defined by giving a list of identifiers after the keyword START. Two groups of identifiers can be separated by the character '-'. The identifiers in the first group are treated as inclusive start states while the identifiers in the second group are treated as exclusive start states. The identifiers in every group may be separated by commas. The standard state STD is predefined as an inclusive start state.

- A pattern given without start states is recognized when the scanner is in any inclusive start state. The exclusive start states are not considered.
- A pattern preceded by the characters '#*#' is recognized when the scanner is in any start state. Both, inclusive and exclusive start states are considered.
- A pattern preceded by a list of start states (enclosed 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. Instead of the keyword NOT the character '-' can be used as well.

Example:

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

The above example shows how to handle nested comments in a Modula-2 scanner. The rule for opening comment brackets is recognized in all inclusive start states. The nesting level is increased and we change the start state to the inclusive start state *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.

Example:

```
START S - T U
RULE
                    A : {}
#*#                 B : {}
#STD#               C : {}
#S#                 D : {}
#T#                 E : {}
#S, T#              F : {}
-#S, T#             G : {}
NOT #U#             H : {}
```

This example declares one inclusive start state S and two exclusive start states T and U. The following table gives for every rule the set of start states where the rule is active.

Table: Start States

Rule	Start States
A	STD, S
B	STD, S, T, U
C	STD
D	S
E	T
F	S, T
G	STD, U
H	STD, S, T

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 and, if the target language requires it, 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.

If the target language is Java, this name may include a package name:

Example:

```
SCANNER mydomain.mypackage.Lexer
```

Here the scanner name is *Lexer* and the generated class will include a package declaration placing it in *mydomain.mypackage*.

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:

- The IMPORT section is used to declare use of other modules by the scanner.
  For Ada, target code after the keyword IMPORT is included in the specification part of the generated scanner before the package header. It can be used to introduce WITH and USE clauses.
  For Java, target code after the keyword IMPORT is included at the head of the class file (after the package declaration if there is one). It may be used to add *import* statements.
  For other languages an IMPORT section is treated like an EXPORT section.
- 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 in order 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 IMPORT, EXPORT, GLOBAL, or DEFAULT sections are not used then the following predefined declarations are included:

If the target language is C:

```
EXPORT {
# include "Position.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 C++:

```
EXPORT {
# include "Position.h"

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

GLOBAL {
void <Scanner>::ErrorAttribute (int Token, <Scanner>_tScanAttribute * Attribute)
   { }
}

DEFAULT {
   yyEcho;
}
```

If the target language is Modula-2:

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

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

DEFAULT {
   yyEcho;
}
```

If the target language is Ada:

```
EXPORT {
type tScanAttribute is record Position: tPosition; end record;
procedure ErrorAttribute (Token: Integer; Attribute: out tScanAttribute);
}

GLOBAL {
procedure ErrorAttribute (Token: Integer; Attribute: out tScanAttribute) is
   begin
      null;
   end ErrorAttribute;
}

DEFAULT {
Text_Io.Put (Text_Io.Standard_Output, yyChBufferPtr (yyChBufferIndex-1));
}
```

If the target language is Eiffel:

```
GLOBAL {
ErrorAttribute (Token: INTEGER): ScanAttribute is
   do
      Result := Attribute;
   end;
}

DEFAULT {
   yyEcho;
}
```

If the target language is Java:

```
IMPORT {
import de.cocolab.reuse.*;
}

EXPORT {
    class ScanAttribute extends Position {
        public ScanAttribute (int line, int column) {
            super (line, column);
        }

        public ScanAttribute (Position other) {
            super (other.line, other.column);
        }
    }

    public ScanAttribute errorAttribute (int token) {
        return new ScanAttribute (Position.NoPosition);
    }
}

DEFAULT {
    yyEcho ();
}
```

These sections import the type tPosition from a module named Position 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 Position 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 in order 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 IMPORT, 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 tString;}
```

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} + :- {}
```

The automatic computation of the line and column position for every token is the default behaviour of a generated scanner. This mechanism can be changed. It is implemented using three *cpp* macros which are predefined as follows:

If the target language is C:

```
# define yyColumn(Ptr) ((int) ((Ptr) - (char *) yyLineStart))
# define yyOffset(Ptr) (yyFileOffset + ((Ptr) - yyChBufferStart2))
# define yySetPosition <Scanner>_Attribute.Position.Line = yyLineCount; \
         <Scanner>_Attribute.Position.Column = yyColumn (<Scanner>_TokenPtr);
```

If the target language is C++:

```
# define yyColumn(Ptr) ((int) ((Ptr) - (char *) yyLineStart))
# define yyOffset(Ptr) (yyFileOffset + ((Ptr) - (char *) yyChBufferStart))
# define yySetPosition Attribute.Position.Line   = yyLineCount; \
                       Attribute.Position.Column = yyColumn (TokenPtr);
```

If the target language is Modula-2:

```
# define yyColumn(Index) ((Index) - yyLineStart)
# define yyOffset(Index) (yyFileOffset + ((Index) - yyChBufferStart))
# define yySetPosition Attribute.Position.Line   := yyLineCount; \
                       Attribute.Position.Column := yyColumn (TokenIndex);
```

If the target language is Ada:

```
# define yyColumn(Index) ((Index) - yyLineStart)
# define yyOffset(Index) (yyFileOffset + ((Index) - yyChBufferStart))
# define yySetPosition Attribute.Position.Line   := yyLineCount; \
                       Attribute.Position.Column := yyColumn (TokenIndex);
```

If the target language is Java:

```
# define yyColumn(Index) ((Index) - yyLineStart)
# define yyOffset(Index) (yyFileOffset + ((Index) - yyChBufferStart))
# define yySetPosition \
   attribute = new ScanAttribute (yyLineCount, tokenIndex - yyLineStart);
```

The macro *yyColumn* determines the column number for a given buffer location (TokenPtr in C and C++, TokenIndex in Modula-2, Java, and Ada). The macro *yySetPosition* is used by the generated scanner in order to assign the position data to the variable Attribute. It can be redefined by the user in the GLOBAL section. This allows for example for the following:

It is possible to get rid of the fields Line and Column.

The fields Line and Column can be named differently.

It is possible to implement a completely different representation for source positions such as e. g. the absolute character offset in a file (as is used by fseek of Unix). This is achieved by using the macro *yyOffset* which determines the offset value for a given buffer location. In the following example the fields *Offset* and *End* receive the absolute character positions of the beginning and the end of a token.

```
# define yySetPosition Attribute.Position.Offset = yyOffset (TokenPtr); \
         Attribute.Position.End = yyOffset (TokenPtr + TokenLength - 1);
```

3.9. Character Set

Scanners generated by Rex depend on the internal representation of the character set. The reason originates from the implementation of the finite automaton which uses in principal a two-dimensional array that maps a state and a character to a new state:

State := Table [State] [Character];

The internal representation of the characters is used for the array access during run time of the scanner as well as during generation time of the table. In order to make a scanner work, these two internal representations have to agree. This is no problem as long as a scanner is generated on a machine with the same encoding of characters as the machine where the scanner is supposed to run on. For example, if both machines use ASCII everything is fine. However, if the encoding of characters is different, then Rex has to know about the internal representation of the character set on the target machine. This can be done by a specification like the following:

```
CHARACTER_SET {
0         0xf0
1         0xf1
...
9         0xf9
A         0xc1
B         0xc2
..
Z         0xe9
a         0x81
b         0x82
...
z         0xa9
0x09      0x05      /* tab         */
\n        0x25      /* newline     */
32        0x40      /* space       */
\\        0xe0      /* back slash */
{         0xc0
|         0x6a
}         0xd0
}
```

The curly brackets after the keyword CHARACTER_SET contain a list of pairs. A pair describes a translation and it consists of a character and its internal code. A character is given by a printable character, a C escape sequence (\n, \t, \v, \b, \r, \f), or a number and a code is given by a number. The numbers can be either decimal, octal, or hexadecimal numbers. Like in C, octal numbers start with the digit ’0’ and hexadecimal numbers with ’0x’. While the character refers to the representation on the host machine the code refers to the representation on the target machine. If no translation is given for a character, then the internal representation of the host machine will be used.

The following should be noted if a character set is specified: The option -i might be necessary if codes greater than 127 are used. (Option -i selects an 8 bit representation for characters.) The action statements <Scanner>_GetLower and <Scanner>_GetUpper (see section 4.4.) may not work because they rely on ASCII. The operator ’<’ for matching the beginning of lines may cause troubles. This feature is implemented by a test whether the character before a line is an end of line character. The end of line character is predefined as the ASCII newline character or whatever this character is translated to by the specification of the character set:

```
# define yyEolCh        (unsigned char) '\12'
```

For C and C++ scanners this definition can be overwritten by supplying an appropriate preprocessor directive in the GLOBAL section.

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 default. 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. The implicit definition of the first 3 rules can be switched off with option -v. The fourth rule can be overwritten using the DEFAULT section.

```
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. The exact syntax varies according to the target language, for example GetWord may be a function returning a value; see the next section for details.

<Scanner>_GetWord (v);

This statement gives access to the matched character sequence.
In C or C++ the sequence is returned in the variable v which must be of type char v [ ] or wchar_t 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.
yyPush (s);	The current start state is pushed on a stack and the start state is changed to state s;
yyPop ();	The start state is changed to the state popped from a stack.
yyPrevious;	The start state is changed to the state valid before the last execution of yyStart, yyPush, yyPop, 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.
input ();	This is a function call returning the next character from the input. It is used where regular expressions alone are not able to describe the input language, for example Fortran style constants.
unput (c);	This pushes the character c back into the input, to be considered when scanning for the next token.

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. The interfaces are discussed in language specific chapters because the syntactic details vary from one target language to another.

5.1. C

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

sions of Rex.

5.1.1. Scanner Interface

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

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

# define          <Scanner>_EofToken        0
# define          <Scanner>_xxMaxCharacter  255

# if xxMaxCharacter < 256
#  define         <Scanner>_xxtChar         char
# else
#  define         <Scanner>_xxtChar         wchar_t
# endif

extern  <Scanner>_xxtChar * <Scanner>_TokenPtr;
extern  int       <Scanner>_TokenLength     ;
extern  <Scanner>_tScanAttribute <Scanner>_Attribute;
extern  void   (* <Scanner>_Exit) (void)    ;

extern  void      <Scanner>_BeginScanner    (void);
extern  void      <Scanner>_BeginFile       (char * FileName);
extern  void      <Scanner>_BeginFileW      (wchar_t * FileName);
extern  void      <Scanner>_BeginMemory     (void * InputPtr);
extern  void      <Scanner>_BeginMemoryN    (void * InputPtr, int Length);
extern  void      <Scanner>_BeginGeneric    (void * InputPtr);
extern  int       <Scanner>_GetToken        (void);
extern  int       <Scanner>_GetWord         (<Scanner>_xxtChar * Word);
extern  int       <Scanner>_GetLower        (<Scanner>_xxtChar * Word);
extern  int       <Scanner>_GetUpper        (<Scanner>_xxtChar * Word);
extern  void      <Scanner>_CloseFile       (void);
extern  void      <Scanner>_CloseScanner    (void);
extern  void      <Scanner>_ResetScanner    (void);
```

- 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 in order to open an input file or a nested include file. It has one parameter of type 'char *' (string) which specifies the file name. The value NULL indicates input from standard input. If not called input is read from standard input. Include files may be nested to arbitrary depth.
- The procedure <Scanner>_BeginFileW does the same as <Scanner>_BeginFile for file names given by wide character strings.
- The procedure <Scanner>_BeginMemory may be called in order to indicate that input should be read from the null terminated string of input items at location InputPtr. The input string may not contain null characters. The contents of the string may not be changed until it has been processed completely.

- The procedure <Scanner>_BeginMemoryN may be called in order to indicate that that the input should be Length input items at location InputPtr. The input may contain null characters. The contents of the input may not be changed until it has been processed completely.
- The procedure <Scanner>_BeginGeneric may be called in order to indicate that the input is user-defined at location InputPtr. The source module (see below) has to be extended by the user in order to implement this feature.
- The procedure <Scanner>_CloseFile may be called in order to close the current input stream (before reaching end of file or end of input). <Scanner>_CloseFile is called automatically by the scanner upon reaching end of file or end of input.
- The procedure <Scanner>_BeginScanner may be called in order to initialize user data. The contents of the target code section named BEGIN is included in the body of this procedure.
- The procedure <Scanner>_CloseScanner may be called in order to finalize user data. The contents of the target code section named CLOSE is included in the body of this procedure.
- If the scanner reaches the end of the input it returns the special token called <Scanner>_EofToken which is encoded by 0.
- The preprocessor symbol <Scanner>_xxMaxCharacter is used to describe the range of the character set.
- The preprocessor symbol <Scanner>_xxtChar is defined to be either char or wchar_t. It describes the type used as representation of a character. Note, the size of wchar_t can be 2 or 4 bytes, depending on the compiler.
- 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 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 current 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 current 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 in order 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 <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 in order to achieve a different behaviour.

The internal scanner interface consists of the following objects:

- The initial size of the scanner input buffer is defined by the value of the preprocessor symbol yyInitBufferSize with a default of 8448. The buffer size is increased automatically when necessary. The initial buffer size can be changed by including a preprocessor directive in the GLOBAL section such as:

```
# define yyInitBufferSize 562
```

  For best results, the value should be a power of two plus a constant between 50 and 256.

- The initial size of the stack for include files is defined by the value of the preprocessor symbol yyInitFileStackSize with a default of 8. The stack size is increased automatically when necessary. The initial stack size can be changed by including a preprocessor directive in the GLOBAL section such as:

```
# define yyInitFileStackSize 16
```

- The value for tab stops is defined by the preprocessor symbol yyTabSpace with a default of 8. This value can be changed by including a preprocessor directive in the GLOBAL section such as:

```
# define yyTabSpace 4
```

5.1.2. Source Interface

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

```
extern void <Scanner>_SetEncoding        (int Encoding, int Endian);
extern int  <Scanner>_BeginSourceFile    (char * FileName);
extern int  <Scanner>_BeginSourceFileW   (wchar_t * FileName);
extern void <Scanner>_BeginSourceMemory  (void * InputPtr);
extern void <Scanner>_BeginSourceMemoryN (void * InputPtr, int Length);
extern void <Scanner>_BeginSourceGeneric (void * InputPtr);
extern int  <Scanner>_GetLine            (int File, char * Buffer, int Size);
extern int  <Scanner>_GetWLine           (int File, wchar_t * Buffer, int Size);
extern void <Scanner>_CloseSource        (int File);
```

- <Scanner>_BeginSourceFile is called from the scanner function <Scanner>_BeginFile indicating that input should be read from a file. The file specified by the parameter FileName is opened and used as input file. If not called input is read from standard input. The function should return an integer file descriptor as provided by the system call open or any other handle understood by the function <Scanner>_GetLine.
- <Scanner>_BeginSourceFileW is called from the scanner function <Scanner>_BeginFileW. It does the same as <Scanner>_BeginSourceFile for file names given by wide character strings. The source module has to be extended by the user in order to implement this feature.
- <Scanner>_BeginSourceMemory is called from the scanner function <Scanner>_BeginMemory indicating that input should be read from the null terminated string of input items at location InputPtr. The input string may not contain null characters. The contents of the string may not be changed until it has been processed completely.

- <Scanner>_BeginSourceMemoryN is called from the scanner function <Scanner>_BeginMemoryN indicating that the input should be Length input items at location InputPtr. The input may contain null characters. The contents of the input may not be changed until it has been processed completely.
- <Scanner>_BeginSourceGeneric is called from the scanner function <Scanner>_BeginGeneric indicating that the input is user-defined at location InputPtr. The source module has to be extended by the user in order to implement this feature.
- <Scanner>_GetLine is called from the scanner in order to fill a buffer at address Buffer with a block of maximal Size characters. Input should be read from a file specified by the integer file descriptor File if the current input stream comes from a file. Otherwise input comes from memory and the parameter File can be ignored. Lines are terminated by newline characters (ASCII = 0xa). The function returns the number of characters transferred. Reasonable block sizes are between 128 and 8192 or the length of a line. Smaller block sizes - especially block size 1 - will drastically slow down the scanner. The end of file or end of input is indicated by a return value <= 0.
- <Scanner>_GetWLine is the same as <Scanner>_GetLine for type wchar_t instead of type char.
- <Scanner>_CloseSource is called from the scanner function <Scanner>_CloseFile at end of file or at end of input, respectively. It can be used to close files. The functions <Scanner>_BeginSource... and <Scanner>_CloseSource can be called in a nested way, for example in order to handle include files. The encoding and the endian property of the input stream are stacked. Therefore after a call of <Scanner>_CloseSource the properties of the previous input stream are restored.
- The function <Scanner>_SetEncoding can be called by the user in order to specify the encoding and the endian property of the input stream. The arguments have to be values as defined below. This function has to be called after the function <Scanner>_BeginSource... . If neither little-endian nor big-endian is specified then the endian property of the current system is assumed to hold for the input. The function <Scanner>_GetWLine will convert the input stream to a stream of type wchar_t.

  The following constants describe the encoding of the input stream:

```
# define CODE_NONE      0
# define CODE_BYTE      1       /* 1 byte         */
# define CODE_WCHAR_T   2       /* 2 or 4 bytes   */
# define CODE_UCS2      3       /* 2 bytes        */
# define CODE_UCS4      4       /* 4 bytes        */
# define CODE_UTF8      5       /* seq of 1 byte  */
# define CODE_UTF16     6       /* seq of 2 bytes */
```

  The above comments give the size of an input stream item in bytes. All input stream items (or sequences of input stream items in the cases of UTF8 and UTF16) represent Unicode characters. The encodings BYTE, UCS2, and UTF16, and possibly WCHAR_T can represent subsets of the full Unicode character set, only. A Unicode character will be stored in variables of type wchar_t. Note, the size of wchar_t can be 2 or 4 bytes, depending on the compiler. Therefore, if the size of wchar_t is 2 then characters encoded by UCS4, UTF8, and UTF16 will be truncated to two bytes.

  The following constants describe the endian property of the input stream:

```
   # define ENDIAN_NONE    0       /* no endian property specified        */
   # define ENDIAN_LITTLE  1       /* little-endian                       */
   # define ENDIAN_BIG     2       /* big-endian                          */
```

5.1.3. Scanner Driver

A main program is necessary for the test of a generated scanner. 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 "Position.h"
# include "<Scanner>.h"

int main (void)
{
   int Token, Count = 0;
   char Word [2048];

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

5.2. C++

5.2.1. Scanner Interface

The scanner interface consists of two parts: While the objects specified in the external interface can be used from outside the scanner, the objects of the internal interface can be used only within a scanner description. The external scanner interface is described by a class named <Scanner>. The name <Scanner> may be specified after the keyword SCANNER. It defaults to *Scanner.* The class definition is contained in a file named <Scanner>.h which has the following contents:

```
# include "Position.h"

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

# define  <Scanner>_xxMaxCharacter  255

# if xxMaxCharacter < 256
#  define <Scanner>_xxtChar         char
# else
#  define <Scanner>_xxtChar         wchar_t
# endif

# define <Scanner>_BASE_CLASS

class <Scanner> <Scanner>_BASE_CLASS {

public:
# define                <Scanner>_EofToken 0
   <Scanner>_xxtChar *  TokenPtr        ;
   int                  TokenLength     ;
   <Scanner>_tScanAttribute Attribute   ;
   void                 (* Exit) (void) ;

                        <Scanner>       (void);
   void                 BeginFile       (char * FileName);
   void                 BeginFileW      (wchar_t * FileName);
   void                 BeginMemory     (void * InpuPtr);
   void                 BeginMemoryN    (void * InpuPtr, int Length);
   void                 BeginGeneric    (void * InpuPtr);
   int                  GetToken        (void);
   int                  GetWord         (<Scanner>_xxtChar * Word);
   int                  GetLower        (<Scanner>_xxtChar * Word);
   int                  GetUpper        (<Scanner>_xxtChar * Word);
   void                 CloseFile       (void);
                        ~<Scanner>      (void);
   void                 ErrorAttribute  (int Token,
                                         <Scanner>_tScanAttribute * Attribute);
   Errors *             ErrorsObj       ;
};
```

- The method 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 method BeginFile may be called in order to open an input file or a nested include file. It has one parameter of type ’char *’ (string) which specifies the file name. The vaue NULL indicates input from standard input. If not called input is read from standard input. Include files may be nested to arbitrary depth.
- The method BeginFileW does the same as BeginFile for file names given by wide character strings.
- The method BeginMemory may be called in order to indicate that input should be read from the null terminated string of input items at location InputPtr. The input string may not contain null characters. The contents of the string may not be changed until it has been processed completely.
- The method BeginMemoryN may be called in order to indicate that that the input should be Length input items at location InputPtr. The input may contain null characters. The contents of the input may not be changed until it has been processed completely.

- The method BeginGeneric may be called in order to indicate that the input is user-defined at location InputPtr. The source module (see below) has to be extended by the user in order to implement this feature.
- The method CloseFile may be called in order to close the current input file (before reaching end of file or end of input). CloseFile is called automatically by the scanner upon reaching end of file or end of input.
- The constructor <Scanner> is called automatically in order to initialize a scanner object. The contents of the target code section named BEGIN is included in the body of this method.
- The destructor ~<Scanner> is called automatically in order to finalize a scanner object. The contents of the target code section named CLOSE is included in the body of this method.
- If the scanner reaches the end of the input it returns the special token called <Scanner>_EofToken which is encoded by 0.
- The preprocessor symbol <Scanner>_xxMaxCharacter is used to describe the range of the character set.
- The preprocessor symbol <Scanner>_xxtChar is defined to be either char or wchar_t. It describes the type used as representation of a character. Note, the size of wchar_t can be 2 or 4 bytes, depending on the compiler.
- The methods GetWord, GetLower, and GetUpper allow access to the matched character sequence as described in section 4.4.
- Alternatively, the matched character sequence can be accessed using the member variables TokenPtr and TokenLength. TokenPtr points to the beginning of the matched character sequence. TokenLength specifies the number of matched characters. Note, the matched character sequence is not terminated by a '\0' character.
- The member variable Attribute is supposed to communicate additional properties of the current 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 current 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 method ErrorAttribute in order to ask for the additional properties of an inserted token which is given by the parameter Token. The method 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 in order to achieve a different behaviour.
- The preprocessor symbol <Scanner>_BASE_CLASS can be used to specify a base class for the class <Scanner> using a #define directive in the EXPORT section of a scanner description. Example:

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

The internal scanner interface consists of the following objects:

- The initial size of the scanner input buffer is defined by the value of the preprocessor symbol yyInitBufferSize with a default of 8448. The buffer size is increased automatically when necessary. The initial buffer size can be changed by including a preprocessor directive in the GLOBAL section such as:

  ```
  # define yyInitBufferSize 562
  ```

  For best results, the value should be a power of two plus a constant between 50 and 256.

- The initial size of the stack for include files is defined by the value of the preprocessor symbol yyInitFileStackSize with a default of 8. The stack size is increased automatically when necessary. The initial stack size can be changed by including a preprocessor directive in the GLOBAL section such as:

  ```
  # define yyInitFileStackSize 16
  ```

- The value for tab stops is defined by the preprocessor symbol yyTabSpace with a default of 8. This value can be changed by including a preprocessor directive in the GLOBAL section such as:

  ```
  # define yyTabSpace 4
  ```

5.2.2. Source Interface

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

```
extern void <Scanner>_SetEncoding        (int Encoding, int Endian);
extern int  <Scanner>_BeginSourceFile    (char * FileName);
extern int  <Scanner>_BeginSourceFileW   (wchar_t * FileName);
extern void <Scanner>_BeginSourceMemory  (void * InputPtr);
extern void <Scanner>_BeginSourceMemoryN (void * InputPtr, int Length);
extern void <Scanner>_BeginSourceGeneric (void * InputPtr);
extern int  <Scanner>_GetLine            (int File, char * Buffer, int Size);
extern int  <Scanner>_GetWLine           (int File, wchar_t * Buffer, int Size);
extern void <Scanner>_CloseSource        (int File);
```

- <Scanner>_BeginSourceFile is called from the scanner method BeginFile indicating that input should be read from a file. The file specified by the parameter FileName is opened and used as input file. If not called input is read from standard input. The function should return an integer file descriptor as provided by the system call open or any other handle understood by the function <Scanner>_GetLine.
- <Scanner>_BeginSourceFileW is called from the scanner method BeginFileW. It does the same as <Scanner>_BeginSourceFile for file names given by wide character strings. The source module has to be extended by the user in order to implement this feature.

- <Scanner>_BeginSourceMemory is called from the scanner method BeginMemory indicating that input should be read from the null terminated string of input items at location InputPtr. The input string may not contain null characters. The contents of the string may not be changed until it has been processed completely.
- <Scanner>_BeginSourceMemoryN is called from the scanner method BeginMemoryN indicating that the input should be Length input items at location InputPtr. The input may contain null characters. The contents of the input may not be changed until it has been processed completely.
- <Scanner>_BeginSourceGeneric is called from the scanner method BeginGeneric indicating that the input is user-defined at location InputPtr. The source module has to be extended by the user in order to implement this feature.
- <Scanner>_GetLine is called from the scanner in order to fill a buffer at address Buffer with a block of maximal Size characters. Input should be read from a file specified by the integer file descriptor File if the current input stream comes from a file. Otherwise input comes from memory and the parameter File can be ignored. Lines are terminated by newline characters (ASCII = 0xa). The function returns the number of characters transferred. Reasonable block sizes are between 128 and 8192 or the length of a line. Smaller block sizes - especially block size 1 - will drastically slow down the scanner. The end of file or end of input is indicated by a return value <= 0.
- <Scanner>_GetWLine is the same as <Scanner>_GetLine for type wchar_t instead of type char.
- <Scanner>_CloseSource is called from the scanner method CloseFile at end of file or at end of input, respectively. It can be used to close files. The functions <Scanner>_BeginSource... and <Scanner>_CloseSource can be called in a nested way, for example in order to handle include files. The encoding and the endian property of the input stream are stacked. Therefore after a call of <Scanner>_CloseSource the properties of the previous input stream are restored.
- The function <Scanner>_SetEncoding can be called by the user in order to specify the encoding and the endian property of the input stream. The arguments have to be values as defined below. This function has to be called after the function <Scanner>_BeginSource... . If neither little-endian nor big-endian is specified then the endian property of the current system is assumed to hold for the input. The function <Scanner>_GetWLine will convert the input stream to a stream of type wchar_t.

  The following constants describe the encoding of the input stream:

```
# define CODE_NONE      0
# define CODE_BYTE      1       /* 1 byte         */
# define CODE_WCHAR_T   2       /* 2 or 4 bytes   */
# define CODE_UCS2      3       /* 2 bytes        */
# define CODE_UCS4      4       /* 4 bytes        */
# define CODE_UTF8      5       /* seq of 1 byte  */
# define CODE_UTF16     6       /* seq of 2 bytes */
```

  The above comments give the size of an input stream item in bytes. All input stream items (or sequences of input stream items in the cases of UTF8 and UTF16) represent Unicode characters. The encodings BYTE, UCS2, and UTF16, and possibly WCHAR_T can represent subsets of the full Unicode character set, only. A Unicode character will be stored in variables of type wchar_t. Note, the size of wchar_t can be 2 or 4 bytes, depending on the compiler.

Therefore, if the size of wchar_t is 2 then characters encoded by UCS4, UTF8, and UTF16 will be truncated to two bytes.

The following constants describe the endian property of the input stream:

```
# define ENDIAN_NONE    0       /* no endian property specified         */
# define ENDIAN_LITTLE  1       /* little-endian                        */
# define ENDIAN_BIG     2       /* big-endian                           */
```

5.2.3. Scanner Driver

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

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

int main (void)
{
   int Token, Count = 0;
   <Scanner> Scanner;

   do {
      Token = Scanner.GetToken ();
      Count ++;
# ifdef Debug
      char Word [2048];
      if (Token != <Scanner>_EofToken) Scanner.GetWord (Word);
      else Word [0] = '\0';
      WritePosition (stdout, Scanner.Attribute.Position);
      printf ("%5d %s\n", Token, Word);
# endif
   } while (Token != <Scanner>_EofToken);
   printf ("%d\n", Count);
   return 0;
}
```

5.3. Modula-2

5.3.1. Scanner Interface

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

```
DEFINITION MODULE <Scanner>;

IMPORT Position, Strings;

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

CONST EofToken  = 0;

VAR TokenLength : INTEGER;
VAR TokenIndex  : INTEGER;
VAR Attribute   : tScanAttribute;
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 procedure BeginFile may be called in order to open an input file or a nested include file. The parameter FileName specifies the file name. The value "" (empty string) denotes input from standard input. If not called input is read from standard input. Include files up to a nesting depth of 15 can be processed.
- The procedure CloseFile may be called in order to close the current 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 in order to initialize user data. The contents of the target code section named BEGIN is included in the body of this procedure.
- The procedure CloseScanner may be called in order to finalize user data. The contents of the target code section named CLOSE is included in the body of this procedure.
- 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 matched characters.
- The variable TokenIndex is an array index of the internal buffer, an array of characters, which specifies the location where the matched character sequence starts. It can be used as argument for the macros that compute source positions.
- The variable Attribute is supposed to communicate additional properties of the current 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 current 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 in order 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 in order 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.

5.3.2. 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 in order 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.

5.3.3. Scanner Driver

A main program is necessary for the test of a generated scanner. 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,
                       TokenLength, CloseScanner;
FROM Strings    IMPORT tString, ArrayToString, WriteL;
FROM IO         IMPORT StdOutput, WriteI, WriteC, WriteNl, CloseIO;
FROM Position   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;
      rExit (0);
   END <Scanner>Drv.
```

5.4. Ada

5.4.1. Scanner Interface

The scanners generated by Rex offer an interface given by the following package contained in the file <Scanner>.ads:

```
with Position, Strings;

package <Scanner> is

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

EofToken        : constant Integer := 0;

TokenLength     : Integer;
TokenIndex      : Integer;
Attribute       : tScanAttribute;

procedure BeginScanner  ;
procedure BeginFile     (FileName: String);
function  GetToken      return Integer;
procedure GetWord       (Word: out Strings.tString);
procedure GetLower      (Word: out Strings.tString);
procedure GetUpper      (Word: out 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 procedure BeginFile may be called in order to open an input file or a nested include file. The parameter FileName specifies the file name. The value "" (empty string) denotes input from standard input. If not called input is read from standard input. Include files up to a nesting depth of 15 can be processed.
- The procedure CloseFile may be called in order to close the current 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 in order to initialize user data. The contents of the target code section named BEGIN is included in the body of this procedure.
- The procedure CloseScanner may be called in order to finalize user data. The contents of the target code section named CLOSE is included in the body of this procedure.

- 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 matched characters.
- The variable TokenIndex is an array index of the internal buffer, an array of characters, which specifies the location where the matched character sequence starts. It can be used as argument for the macros that compute source positions.
- The variable Attribute is supposed to communicate additional properties of the current 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 current 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 in order 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.
- If the scanner reaches the end of the input it returns the special token called EofToken which is encoded by 0.

5.4.2. 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.ads and <Scanner>source.adb. The package module in the file <Scanner>source.ads has the following contents:

```
package <Scanner>Source is

function  BeginSource (FileName: String) return Integer;
procedure GetLine     (File: Integer; Buffer: out String; Size: Integer;
                       Last: out Integer);
procedure CloseSource (File: Integer);

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 in order 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.adb has the following contents:

```
with System; Use System;

package body <Scanner>Source is

function BeginSource (FileName: String) return Integer is
      function OpenInput (FileName: Address) return Integer;
      pragma Interface (C, OpenInput);
      pragma Interface_Name (OpenInput, "OpenInput");
      C_Name    : String (1 .. 256);
   begin
      C_Name (1 .. FileName'Last) := FileName;
      C_Name (FileName'Last + 1)  := Character'Val (0);
      return OpenInput (C_Name'Address);
   end BeginSource;

procedure GetLine (File: Integer; Buffer: out String; Size: Integer;
                   Last: out Integer) is
      function rRead (File: Integer; Buffer: Address; Size: Integer)
                      return Integer;
      pragma Interface (C, rRead);
      pragma Interface_Name (rRead, "rRead");
   begin
      Last := rRead (File, Buffer'Address, Size);
   end GetLine;

procedure CloseSource (File: Integer) is
      procedure rClose (File: Integer);
      pragma Interface (C, rClose);
      pragma Interface_Name (rClose, "rClose");
   begin
      rClose (File);
   end CloseSource;

end <Scanner>Source;
```

5.4.3. Scanner Driver

A main program is necessary for the test of a generated scanner. Rex can provide a minimal main program in the file <Scanner>drv.adb which can serve as test driver. It counts the tokens and looks like the following:

```
with <Scanner>, Text_Io, Position, Strings;
use  <Scanner>, Text_Io, Position, Strings;

procedure <Scanner>Drv is

   package Int_Io is new Text_Io.Integer_IO (Integer); use Int_Io;

   Token        : Integer       := 1;
   Word         : tString;
   Debug        : Boolean       := False;
   Count        : Integer       := 0;
begin
   BeginScanner;
   while Token /= EofToken loop
      Token := GetToken;
      Count := Count + 1;
      if Debug then
         WritePosition (Standard_Output, Attribute.Position);
         Put (Standard_Output, Token, 5);
```

```
         if TokenLength > 0 then
            Put (Standard_Output, ' ');
            GetWord (Word);
            WriteS (Standard_Output, Word);
         end if;
         New_Line (Standard_Output);
      end if;
   end loop;
   CloseScanner;
   Put (Standard_Output, Count, 0);
   New_Line (Standard_Output);
end <Scanner>Drv;
```

5.5. Eiffel

5.5.1. Scanner Interface

The file <Scanner>.e contains the class <Scanner> which offers the following features:

```
class <Scanner>

creation BeginScanner

feature

EofToken        : INTEGER is 0
TokenLength     : INTEGER
Attribute       : ScanAttribute

BeginScanner    is
BeginFile       (FileName: STRING) is
GetToken        : INTEGER is
GetWord         : STRING is
GetLower        : STRING is
GetUpper        : STRING is
CloseFile       is
CloseScanner    is

ErrorAttribute  (Token: INTEGER): ScanAttribute is
SetAttribute    (Value: ScanAttribute) is
```

- 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 procedure BeginFile may be called in order to open an input file or a nested include file. The parameter FileName specifies the file name. The value "" (empty string) denotes input from standard input. If not called input is read from standard input. Include files may be nested to arbitrary depth.
- The procedure CloseFile may be called in order to close the current input file (before reaching end of file). CloseFile is called automatically by the scanner upon reaching end of file.
- The procedure BeginScanner instantiates a scanner object and performs the necessary initializations. For example, the tables are read in from a file named <scanner>.txt. The contents of the target code section named BEGIN is included in the body of this procedure.
- The procedure CloseScanner may be called in order to finalize user data. The contents of the target code section named CLOSE is included in the body of this procedure.

- 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 matched characters.
- The variable Attribute is supposed to communicate additional properties of the current token. The value must be provided by appropriate action statements. The class of this feature has to be a subclass of the predefined support class ScanAttribute. This class has one feature called Position of type Position. The class Position has two features called Line and Column. The values of Line and Column are computed automatically by the scanner. They indicate the source position of the current token. The position of a token is the position of the first character of the token. For exceptions see section 3.8. The classes ScanAttribute and Position are predefined in the library reuse/eiffel. Subclasses of these classes can be defined in order reflect application specific needs.
- During automatic error repair a parser may insert tokens. In this case the parser calls the procedure ErrorAttribute in order to ask for the additional properties of an inserted token which is given by the parameter Token. The procedure should return a default value for the additional properties of the token Token.
- The procedure SetAttribute can be used to store values in the variable Attribute.
- If the scanner reaches the end of the input it returns the special token called EofToken which is encoded by 0.

5.5.2. Source Interface

The scanners generated by Rex need a source class for blocked input of characters. Rex can provide a prototype source module which reads from standard input. It is contained in the file source.e and has the following interface:

```
class Source

creation Open

feature

Open (filename: STRING) is
GetLine (wanted: INTEGER): STRING is
Close is
```

- Open 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 in order to return a block of maximal 'wanted' characters.
- Close is called from the scanner at end of file respectively at end of input. It can be used to close files.

5.5.3. Scanner Driver

A main program is necessary for the test of a generated scanner. Rex can provide a minimal main program in the file <Scanner>drv.e which can serve as test driver. It counts the tokens and looks as follows:

```
class <Scanner>Drv

creation main

feature
```

```
main is
   local
      Token     : INTEGER
      Count     : INTEGER
      Scanner   : <Scanner>
      f         : rFILE
      Attribute : ScanAttribute
   do
      !! f.make_write_from_fp (f.stdout_fp)
      !! Scanner.BeginScanner
      from
         Token := Scanner.GetToken
         Count := 1
         debug
            Scanner.Attribute.Position.WritePosition (f)
            f.putint2 (Token, 5)
            f.putchar (' ')
            f.putstring (Scanner.GetWord)
            f.new_line
         end
      until Token = Scanner.EofToken loop
         Token := Scanner.GetToken
         Count := Count + 1
         debug
            Scanner.Attribute.Position.WritePosition (f)
            f.putint2 (Token, 5)
            f.putchar (' ')
            f.putstring (Scanner.GetWord)
            f.new_line
         end
      end
      Scanner.CloseScanner
      f.putint (Count)
      f.new_line
      f.close
   end
end
```

5.6. Java

5.6.1. Scanner Interface

The file <Scanner>.java contains the class <Scanner> which offers the following features as default:

```
import de.cocolab.reuse.*;

public class <Scanner> {

    class ScanAttribute implements HasPosition {
    }

    public ScanAttribute errorAttribute (int token) {
    }

    public static final int eofToken    = 0;

    public int          tokenLength;

    public ScanAttribute attribute;
```

```
    public              <Scanner>       () throws java.io.IOException;
    public void         beginFile       (java.io.InputStream stream)
                                           throws java.io.IOException;
    public int          getToken        () throws java.io.IOException;
    public String       getWord         ();
    public String       getLower        ();
    public String       getUpper        ();
    public void         closeFile       () throws java.io.IOException;
    public void         finalize        ();
}
```

- 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 procedure *beginFile* may be called in order to open an input file or a nested include file. The parameter *stream* specifies the input source. If not called input is read from standard input. Include files may be nested to arbitrary depth.
- The procedure *closeFile* may be called in order to close the current input file (before reaching end of file). *closeFile* is called automatically by the scanner upon reaching end of file.
- The contents of the target code section named BEGIN is included in the contructor <Scanner>(), and is executed whenever a new scanner object is created.
- The procedure *finalize* may be called in order to finalize user data. The contents of the target code section named CLOSE is included in the body of this procedure. A good JVM will call this procedure when the scanner object is garbage collected, or it may be called explicitly.
- 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 matched characters.
- The variable *attribute* is supposed to communicate additional properties of the current token. The value must be provided by appropriate action statements. This variable is of type ScanAttribute which has to be a class which implements the interface HasPosition, i.e. it has a method *position ()* which returns an instance of Position. Position has to be a class with at least two fields called *line* and *column*. This arrangment leaves the user free to decide whether to have a field of type Position or to inherit directly from Position. The default definition of the macro yySetPosition assumes the latter; this minimises the number of objects created. The values of line and column are computed by the scanner, automatically. They indicate the source position of the current token. The position of a token is the position of the first character of the token. For exceptions see section 3.8.
- During automatic error repair a parser may insert tokens. In this case the parser calls the procedure *errorAttribute* in order to ask for the additional properties of an inserted token which is given by the parameter *token*. The procedure should return default values for the additional properties of the token.
- In the event of an internal error in the scanner an exception de.cocolab.reuse.FatalError will be thrown. It is not required to catch this exception.
- If the scanner reaches the end of the input it returns the special token called *eofToken* which is encoded by 0.

The internal scanner interface consists of the following objects:

- The initial size of the scanner input buffer is defined by the value of the preprocessor symbol yyInitBufferSize with a default of 8448. The buffer size is increased automatically when necessary. The initial buffer size can be changed by including a C preprocessor directive in the GLOBAL section such as:

```
# define yyInitBufferSize 562
```

  For best results, the value should be a power of two plus a constant between 50 and 256.

- The stack for include files is supplied by default with unlimited size. If nested include files are not required the size of the generated scanner can be reduced by including a preprocessor directive in the GLOBAL section such as:

```
# define yyInitFileStackSize 0
```

- The value for tab stops is defined by the preprocessor symbol yyTabSpace with a default of 8. This value can be changed by including a preprocessor directive in the GLOBAL section such as:

```
# define yyTabSpace 4
```

- The stack used by yyPush and yyPop is initially 16 elements, and will grow as required. A different initial size can be specified by including a preprocessor directive in the GLOBAL section such as:

```
# define yyInitStStStackSize 32
```

  If the initial size is given as zero then there is no start state stack, and yyPush/yyPop may not be used. This feature may be used to obtain the smallest possible scanner.

5.6.2. Tuning the Scanner Interface

It is not possible to design one interface to the scanner that is optimal in all circumstances. This is because different cases will require different emphasis on the following characteristics:

- the number of objects created per token; there will always be an object representing source location (Position) and there will often be an additional object ecapsulating this together with other information about the token, for example a coded identifier. This is important where the input is large and run time is to be minimized.
- memory usage; the size of an instance of ScanAttribute is important if it is to be stored in an abstract syntax tree and the size of the input may be large.
- the number of classes generated; the time to load a Java applet over the WWW increases with the number of classes. For small input load time is the most significant factor.

A scanner generated for Java may be tuned in a number of ways using macros defined in the GLOBAL section and by proper choice of design for the ScanAttribute class. These facilities are introduced here: more details may be found by examining the skeleton from which Rex generates a Java scanner, i.e. the file Scanner.java found in the lib/rex directory within the Cocktail installation.

- The name ScanAttribute may be defined to Position so that attributes consist directly of an instance of Position (this being the minimum requirement of a *lark* generated parser). This is suitable for a small, fast scanner which does not have to deliver additional attributes.

- ScanAttribute may extend Position instead of having a field of type Position. This avoids an additional object creation at the cost of having some information in the ScanAttribute class about the implementation of Position. Specifically, there needs to be a constructor which mirrors that of Position and a decision must be made as to what *toString ()* should return - just the position or some representation of any additional attributes. For an example of this technique see the default EXPORT section in the skeleton.
- Another way of achieving the same end is to have ScanAttribute implement HasPosition. The values of line and column are stored as fields and used to create an instance of Position only when the *position ()* method is called, that is only if a syntax error is detected. This achieves the aim of creating only one object per token for correct input while avoiding the issue of what *toString ()* should return.
- By default the macro *yySetPosition* is defined to create an instance of ScanAttribute from the position information. This macro is called whenever a pattern is matched and position calculation has not been suppressed (see section 3.8) but *before* any user action code is entered. If the user action code may create some subclass of ScanAttribute in order to include attributes specific to the type of token (the value of a numeric literal for example) then either yySetPosition must be redefined or position calculation must be suppressed for those rules which will instantiate some descendant of ScanAttribute.
- The macro *yyGetTokenBegin* may be used to execute code at the beginning of *getToken ()*, that is for every token read. By default this macro is empty.
- The macro *yyAttributePosition (attribute)* may be used to change how position information is obtained from an instance of ScanAttribute. This is only of significance when the scanner is reporting some internal error such as misuse of the *yyPush/yyPop* methods. A scanner to be used by a *lark* generated parser has other requirements.

5.6.3. Source Interface

The scanners generated by Rex need a source module for blocked input of characters. For Java this is any class which descends from java.io.InputStream.

5.6.4. Scanner Driver

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

```
import java.io.*;

/**
 * Simple class for driving a generated scanner.
 */

class <Scanner>Drv {

    public static void main (String argv []) throws java.io.IOException {
        <Scanner> scanner = new <Scanner> ();
        int token;
        boolean debug = false;
        String filename = null;
        int count = 0;

        for (int i = 0; i < argv.length; i ++) {
            if (argv [i].equals ("-D")) debug = true;
```

```
            else filename = argv [i];
        }
        if (filename != null)
            scanner.beginFile (new FileInputStream (filename));
        do {
            token = scanner.getToken ();
            count ++;
            if (debug) {
                String word = scanner.getWord ();
                System.err.println (scanner.attribute.position () + " " +
                    token + " " + word);
            }
        } while (token != <Scanner>.eofToken);
        scanner.finalize ();
        System.out.println (count);
    }
}
```

6. Usage

NAME

rex − generator of lexical analyzers

SYNOPSIS

rex [-options] [-k{124}] [-f*file*] [-l*directory*] [*file*]

DESCRIPTION

Rex generates program code to be used in lexical analysis of text. A typical application is the generation of scanners for compilers. The generated scanners can handle single byte input as well as Unicode input. The input *file* contains regular expressions to be searched for, and actions written in the implementation language to be executed when strings according to the expressions are found. Unrecognized portions of the input are copied to standard output. In order 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 every 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 such as blanks, tabs, or newlines and there is a mechanism to handle include files. The generated scanners are implemented as table-driven deterministic finite automatons.

OPTIONS

a generate all (= ds)

c generate a lexical analyzer in C

\+ generate a lexical analyzer in C++

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

u generate a lexical analyzer in Ada

e generate a lexical analyzer in Eiffel

j generate a lexical analyzer in Java

d generate a header file or definition module

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

v do not predefine rules for skipping of white space

x require explicit definitions for used identifiers
(default: undefined identifiers are treated as strings)

y do not generate dummy labels (might cause compiler messages such as 'statement not reached') (default: generate dummy labels, might cause compiler messages such as 'label not used'.)

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)

k<n>
generate scanner for characters having n bytes (default: 1) (n > 1 implies -z and disables CHARACTER_SET)

z[<n>]
map characters to classes at run time, use an array of n elements, n >= 256 (default: 16384)

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 character set into blocks during generation (implies -k1)

t touch output files only if necessary

p print information about ambiguous rules

1 print statistics about the generated lexical analyzer

h print help information

-f*file* specify a file to be used as skeleton for the scanner

-l*dir* specify the directory dir where rex finds its data files

FILES

if output is in C:

<Scanner>.h	header file of the generated scanner
<Scanner>.c	body of the generated scanner
<Scanner>Source.h	header file of support module source
<Scanner>Source.c	body of support module source
<Scanner>Drv.c	body of the scanner driver (main program)

if output is in C++:

<Scanner>.h	header file of the generated scanner
<Scanner>.cxx	body of the generated scanner
<Scanner>Source.h	header file of support module source
<Scanner>Source.cxx	body of support module source
<Scanner>Drv.cxx	body of the scanner driver (main program)

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	implementation module of the scanner driver

if output is in Ada:

<Scanner>.ads	package of the generated scanner
<Scanner>.adb	package body of the generated scanner
<Scanner>source.ads	package of support module source
<Scanner>source.adb	package body of support module source
<Scanner>drv.adb	package body of the scanner driver

if output is in Eiffel:

<Scanner>.e	class of the generated scanner
<Scanner>buffer.e	class of the character buffer for the scanner
<Scanner>drv.e	class of the scanner driver (main program)
<Scanner>.txt	tables controlling the generated scanner (ASCII format)
source.e	support class for input
attribute.e	support class for the description of properties of nonterminals
scanattribute.e	support class for the description of properties of tokens
position.e	support class for the representation of source positions
rfile.e	support class extending the class FILE
rsystem.e	support class for system specific properties

if output is in Java:

<Scanner>.java	class of the generated scanner
<Scanner>Drv.java	class of the scanner driver (main program)

SEE ALSO

J. Grosch: "Rex - A Scanner Generator", CoCoLab Germany, Document No. 5

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

J. Grosch: "Efficient Generation of Table-Driven Scanners", CoCoLab Germany, Document No. 2

7. Implementation

Rex is implemented by a 12,000 line Modula-2 program. The program makes heavy use of a library of reusable Modula-2 modules currently comprising 9,000 lines of code [Groa, Grob]. Of the 12,000 lines of Rex around 4,900 lines are generated by tools:

- 2100 lines for the scanner are generated by Rex itself.
- 1500 lines for the parser are generated by the LALR(1) parser generator *lark.*
- 1100 lines for a tree data structure are generated by the abstract syntax tree tool *ast.*
- 250 lines for an attribute evaluator are generated by the attribute evaluator generator *ag.*

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):

Advantages of Rex:

+ Rex can generate scanners for Unicode.
+ 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.

Table: Syntactical differences between Rex and LEX:

Meaning	LEX	Rex
delimiter for character classes	[ ]	{ }
complement of character classes	[^]	- { }
any character	.	ANY
left justification	^	<
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 ( )
character set	%T	CHARACTER_SET
action at end of input	yywrap	EOF section

+ 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 with an unlimited nesting depth 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.

Appendix 1: Syntax of the Specification Language

```
specification   : decls rules
                .
decls           :
                | decls name
                | decls code
                | decls characterSet
                | decls define
                | decls start
                .
name            : SCANNER [Ident]
                | SCANNER DottedIdent /* Java only */
                .
code            : IMPORT  TargetCode
                | EXPORT  TargetCode
                | GLOBAL  TargetCode
                | LOCAL   TargetCode
                | BEGIN   TargetCode
                | CLOSE   TargetCode
                | DEFAULT TargetCode
                | EOF     TargetCode
                .
characterSet    : CHARACTER_SET '{' charDef * '}'
                .
define          : DEFINE definition *
                .
start           : START [identList] ['-' identList]
                .
rules           : RULE  rule *
                | RULES rule *
                .
charDef         : CsChar   CsNumber
                | CsNumber CsNumber
                .
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 '#'
                | '-' '#' identList '#'
                .
regExpr         : regExpr '|' regTerm
                | regTerm
```

```
                .
regTerm         : regTerm regFactor                                          |
                | regFactor                                                  |
                .                                                            |
regFactor       : regFactor '+'                                              |
                | regFactor '*'                                              |
                | regFactor '?'                                              |
                | regFactor '[' Number ']'                                   |
                | regFactor '[' Number '-' Number ']'                        |
                | '(' regExpr ')'
                | charSet
                | Char
                | Ident
                | String
                | Number
                .
charSet         : '-' charSet
                | '{' range * '}'
                .
range           : Char
                | Char '-' Char
                .
Char            : CsChar
                | '\' decimal_number
                | '\' hex_number
                | '\' ('x' | 'X') hex_digit *
                | '\' ('u' | 'U') hex_digit *
                .
Ident           : letter letter_or_digit *
                .
DottedIdent     : Ident
                | DottedIdent '.' Ident
                .
letter_or_digit : letter
                | digit
                | '_'
                .
String          : '"' character * '"'
                .
Number          : decimal_number
                .
Target_code     : '{' character * '}'
                .
CsChar          : character
                | '\' a                                                      |
                | '\' n
                | '\' t
                | '\' v
                | '\' b
                | '\' r
                | '\' f
                | '\' character
                .
CsNumber        : octal_number
                | decimal_number
                | hex_number
                .
```

```
octal_number    : '0' octal_digit *
                .
decimal_number  : digit +
                .
hex_number      : '0' ('x' | 'X') hex_digit *
                .
```

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

```
GLOBAL  {
# include "Memory.h"
# include "StringM.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 StringM(emory).
      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 StringM         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 StringM    IMPORT tStringRef;
FROM Texts      IMPORT tText    ;
FROM Position   IMPORT tPosition;
FROM UniCode    IMPORT UCHAR    ;

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

VAR ErrorCount  : CARDINAL;

PROCEDURE ErrorAttribute (Token: INTEGER; VAR Attribute: tScanAttribute);
PROCEDURE startCode     ;
PROCEDURE startCharset  ;
PROCEDURE startSet      ;
PROCEDURE startRules    ;
}

GLOBAL  {

FROM Strings    IMPORT tString, Concatenate, Char, SubString, cMaxStrLength,
                       StringToInt, StringToNumber, AssignEmpty, Length,
                       ArrayToString, IntToString;
FROM Texts      IMPORT MakeText, Append;
FROM StringM    IMPORT tStringRef, PutString;
FROM Idents     IMPORT tIdent, MakeIdent, NoIdent, GetString;
FROM Errors     IMPORT Message, Error, Restriction;
FROM ScanGen    IMPORT Language, tLanguage, Procedures, AppendLine,
                       pGetWord, pGetLower, pGetUpper, pinput, pyyPush, pyyPop;
FROM Position   IMPORT tPosition;
FROM UniCode    IMPORT MaxUCHAR;

CONST
   SymIdent             = 1     ;
   SymNumber            = 2     ;
   SymString            = 3     ;
   SymChar              = 4     ;
   SymTargetcode        = 5     ;
   SymScanner           = 37    ;
   SymImport            = 39    ;
   SymExport            = 32    ;
   SymGlobal            = 6     ;
   SymLocal             = 31    ;
```

```
   SymBegin             = 7     ;
   SymClose             = 8     ;
   SymEof               = 34    ;
   SymDefault           = 36    ;
   SymCharSet           = 38    ;
   SymDefine            = 9     ;
   SymStart             = 10    ;
   SymRules             = 11    ;
   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    ;
   SymRule              = 30    ;

VAR
   level        : INTEGER       ;
   string, TargetCode   : tString       ;
   NoString     : tStringRef    ;
   StartPosition, StringPosition: tPosition     ;
   PrevState    : SHORTCARD     ;
   IsChar       : BOOLEAN       ;

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

PROCEDURE startCode;
  BEGIN
    yyStart (targetcode);
    MakeText (Attribute.Text);
    AssignEmpty (TargetCode);
    level := 1;
```

```
  END startCode;

PROCEDURE startCharset;
  BEGIN
    yyStart (charset);
    IsChar := TRUE;
  END startCharset;

PROCEDURE startSet;
  BEGIN
    yyStart (set);
  END startSet;

PROCEDURE startRules;
  BEGIN
    yyStart (rules);
  END startRules;
}

LOCAL   {
VAR
   String, Word : tString;
   n, PPLine    : LONGCARD;

PROCEDURE AppendCode;
   BEGIN
      GetWord (Word);
      Concatenate (TargetCode, Word);
   END AppendCode;
}

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

DEFAULT {
   Message ("illegal character", Error, Attribute.Position);
   INC (ErrorCount);
}

EOF     {
   CASE yyStartState OF
   | targetcode ,
     set        : Message ("terminating '}' missing", Error, StartPosition);
                  INC (ErrorCount);
   | comment    : Message ("unterminated comment", Error, StartPosition);
                  INC (ErrorCount);
   | CStr1, CStr2,
     AStr1, AStr2,
     Str1, Str2 : Message ("unterminated string", Error, StringPosition);
                  INC (ErrorCount);
   ELSE
   END;
   yyStart (STD);
```

```
}

DEFINE
   letter       = {A-Z a-z}     .
   digit        = {0-9}         .
   octdigit     = {0-7}         .
   hexdigit     = {0-9 A-F a-f} .
   string       = - {"\n}       .
   cmtch        = - {*\t\n}     .
   code         = - {{\}\t\n\r\\'"gGiy#} .
   StrCh1       = - {'\t\n}     .
   StrCh2       = - {"\t\n}     .
   CStrCh1      = - {'\t\n\\}   .
   CStrCh2      = - {"\t\n\\}   .
   AStrCh       = {\ !#-~}      .

START targetcode, set, rules, comment,
      Str1, Str2, CStr1, CStr2, AStr1, AStr2, charset, PPline

RULES

#targetcode#    "{"     :- {
                           AppendCode;
                           INC (level);
                        }

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

#targetcode#    code + | {gGiy\r#} :- { AppendCode; }

#targetcode#    GetWord :- {IF Language # Java THEN
                              INCL (Procedures, pGetWord);
                            END; AppendCode; }
#targetcode#    getWord :- {IF Language = Java THEN
                              INCL (Procedures, pGetWord);
                            END; AppendCode; }
#targetcode#    GetLower:- {IF Language # Java THEN
                              INCL (Procedures, pGetLower);
                            END;  AppendCode; }
#targetcode#    getLower:- {IF Language = Java THEN
                              INCL (Procedures, pGetLower);
                            END;  AppendCode; }
#targetcode#    GetUpper:- {IF Language # Java THEN
                              INCL (Procedures, pGetUpper);
                            END;  AppendCode; }
#targetcode#    getUpper:- {IF Language = Java THEN
                              INCL (Procedures, pGetUpper);
```

```
                          END;  AppendCode; }

#targetcode#    input   :- { INCL (Procedures, pinput   ); AppendCode; }
#targetcode#    yyPush  :- { INCL (Procedures, pyyPush  ); AppendCode; }
#targetcode#    yyPop   :- { INCL (Procedures, pyyPop   ); AppendCode; }

#targetcode#    \t      :- {
                           Strings.Append (TargetCode, 11C);
                           yyTab;
                        }

#targetcode#    \r ? \n :- {
                           Append (Attribute.Text, TargetCode);
                           AssignEmpty (TargetCode);
                           yyEol (0);
                        }

#targetcode#    \\ ANY  :- {
                           GetWord (Word);
                           Strings.Append (TargetCode, Char (Word, 2));
                        }

#targetcode#    \\ / \r ? \n,
#targetcode#    \\      :- { Strings.Append (TargetCode, '\'); }

#targetcode#    '       : {
                   GetWord (String);
                   IF (Language = C) OR (Language = Cpp) THEN yyStart (CStr1);
                   ELSIF Language = Ada THEN yyStart (AStr1);
                   ELSE yyStart (Str1);
                   END;
                   StringPosition := Attribute.Position;
                        }

#targetcode#    \"      : {
                   GetWord (String);
                   IF (Language = C) OR (Language = Cpp) THEN yyStart (CStr2);
                   ELSIF Language = Ada THEN yyStart (AStr2);
                   ELSE yyStart (Str2);
                   END;
                   StringPosition := Attribute.Position;
                        }

#Str1#  StrCh1 +        ,
#Str2#  StrCh2 +        ,
#CStr1# CStrCh1 + | \\ ANY ? ,
#CStr2# CStrCh2 + | \\ ANY ? ,
#AStr2# AStrCh +        :- {GetWord (Word); Concatenate (String, Word);}

#CStr1, CStr2# \\ \r ? \n:- {Strings.Append (String, '´);
                             Strings.Append (String, 12C); yyEol (0);}

#Str1, CStr1# '         ,
#Str2, CStr2, AStr2# \" ,
#AStr1# ANY ' ?         :- {GetWord (Word); Concatenate (String, Word);
                            yyPrevious; Concatenate (TargetCode, String);
                        }
```

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

#Str1, Str2, CStr1, CStr2, AStr2# \r ? \n :- {
                 (* Message ("unterminated string", Error, Attribute.Position);
                    INC (ErrorCount); *)
                    Strings.Append (String, 12C);
                    yyEol (0); yyPrevious; Concatenate (TargetCode, String);
                        }

#Str1, Str2, CStr1, CStr2# \r :- {Strings.Append (String, 15C);}

#STD, rules, charset#
          "/*"          :  {yyStart (comment);
                            StartPosition := Attribute.Position; }
#comment# "*" | cmtch + :- {}
#comment# "*/"          :- {yyPrevious;}

#STD, rules, charset# "//" ANY * :- {}

#STD# IMPORT            : {PrevState := STD; RETURN SymImport   ;}
#STD# EXPORT            : {PrevState := STD; RETURN SymExport   ;}
#STD# GLOBAL            : {PrevState := STD; RETURN SymGlobal   ;}
#STD# LOCAL             : {PrevState := STD; RETURN SymLocal    ;}
#STD# BEGIN             : {PrevState := STD; RETURN SymBegin    ;}
#STD# CLOSE             : {PrevState := STD; RETURN SymClose    ;}
#STD# DEFAULT           : {PrevState := STD; RETURN SymDefault  ;}
#STD# EOF               : {PrevState := STD; RETURN SymEof      ;}
#STD# SCANNER           : {RETURN SymScanner    ;}
#STD# CHARACTER_SET     : {RETURN SymCharSet    ;}
#STD# DEFINE            : {RETURN SymDefine     ;}
#STD# START             : {RETURN SymStart      ;}
#STD# RULE              : {RETURN SymRule       ;}
#STD# RULES             : {RETURN SymRules      ;}

#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 * \" : {
   IF TokenLength > cMaxStrLength THEN
      Message ("string too long (max. 255)", Restriction, Attribute.Position);
      INC (ErrorCount);
      ArrayToString (" ", TargetCode);
   ELSIF TokenLength = 2 THEN
      Message ("string length > 0 required", Error, Attribute.Position);
      INC (ErrorCount);
      ArrayToString (" ", TargetCode);
   ELSE
      GetWord (Word);
```

```
      SubString (Word, 2, Length (Word) - 1, TargetCode);
   END;
   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#    "{"     : {StartPosition := Attribute.Position;
                           RETURN SymLBrace     ;}
#rules#         "#"     : {RETURN SymNrSign     ;}
#rules#         "/"     : {RETURN SymSlash      ;}
#rules#         "<"     : {RETURN SymLess       ;}
#rules#         ">"     : {RETURN SymGreater    ;}
#rules#         ":"     : {PrevState := rules; RETURN SymColon      ;}
#rules#         ":-"    : {PrevState := rules; RETURN SymColonMinus ;}

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

#STD, set, rules# \\ digit + : {
                   GetWord (Word);
                   SubString (Word, 2, Length (Word), TargetCode);
                   n := LONGCARD (StringToInt (TargetCode));
                   IF n <= MaxUCHAR THEN
                      Attribute.Ch := n;
                   ELSE
                      Message ("number out of range", Error, Attribute.Position);
                      INC (ErrorCount);
                      Attribute.Ch := 0;
                   END;
                   RETURN SymChar;
                        }

#STD, set, rules# \\ "0" {xX} hexdigit + : {
                   GetWord (Word);
                   SubString (Word, 4, Length (Word), TargetCode);
                   n := StringToNumber (TargetCode, 16);
                   IF n <= MaxUCHAR THEN
                      Attribute.Ch := n;
                   ELSE
```

```
                     Message ("number out of range", Error, Attribute.Position);
                     INC (ErrorCount);
                     Attribute.Ch := 0;
                  END;
                  RETURN SymChar;
                       }

#STD, set, rules# \\ {xXuU} hexdigit + : {
                  GetWord (Word);
                  SubString (Word, 3, Length (Word), TargetCode);
                  n := StringToNumber (TargetCode, 16);
                  IF n <= MaxUCHAR THEN
                     Attribute.Ch := n;
                  ELSE
                     Message ("number out of range", Error, Attribute.Position);
                     INC (ErrorCount);
                     Attribute.Ch := 0;
                  END;
                  RETURN SymChar;
                       }

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

#STD, set, rules# \\ / \r \n : {
                          Attribute.Ch := ORD ('\');
                          RETURN SymChar;
                       }

#STD, set, rules# - {\ \t\n\f\r\26} : {
                          GetWord (Word);
                          Attribute.Ch := ORD (Char (Word, 1));
                          RETURN SymChar;
                       }

\f , \r , \26          :- {}

#charset# digit        : {
                          IsChar := NOT IsChar;
                          GetWord (Word);
                          IF NOT IsChar THEN
                             Attribute.Ch := ORD (Char (Word, 1));
                             RETURN SymChar;
                          ELSE
                             Attribute.Number := StringToInt (Word);
                             RETURN SymNumber;
                          END;
                       }

#charset# "0" octdigit *: {
                          IsChar := NOT IsChar;
                          GetWord (Word);
                          Attribute.Number := StringToNumber (Word, 8);
                          RETURN SymNumber;
```

```
                        }

#charset# digit +       : {
                           IsChar := NOT IsChar;
                           GetWord (Word);
                           Attribute.Number := StringToInt (Word);
                           RETURN SymNumber;
                        }

#charset# "0" {xX} hexdigit + : {
                           IsChar := NOT IsChar;
                           GetWord (Word);
                           SubString (Word, 3, Length (Word), TargetCode);
                           Attribute.Number := StringToNumber (TargetCode, 16);
                           RETURN SymNumber;
                        }

#charset# \\ a : {Attribute.Ch := ORD (007C); IsChar := FALSE; RETURN SymChar;}
#charset# \\ b : {Attribute.Ch := ORD (010C); IsChar := FALSE; RETURN SymChar;}
#charset# \\ t : {Attribute.Ch := ORD (011C); IsChar := FALSE; RETURN SymChar;}
#charset# \\ n : {Attribute.Ch := ORD (012C); IsChar := FALSE; RETURN SymChar;}
#charset# \\ v : {Attribute.Ch := ORD (013C); IsChar := FALSE; RETURN SymChar;}
#charset# \\ f : {Attribute.Ch := ORD (014C); IsChar := FALSE; RETURN SymChar;}
#charset# \\ r : {Attribute.Ch := ORD (015C); IsChar := FALSE; RETURN SymChar;}

#charset# \\ ANY        : {
                           IsChar := FALSE;
                           GetWord (Word);
                           Attribute.Ch := ORD (Char (Word, 2));
                           RETURN SymChar;
                        }

#charset# - {\ \t\n\f\r\26\}} : {
                           IsChar := FALSE;
                           GetWord (Word);
                           Attribute.Ch := ORD (Char (Word, 1));
                           RETURN SymChar;
                        }

#charset# "}" / {\ \t\n} * digit : {
                           IsChar := FALSE;
                           Attribute.Ch := ORD ('}');
                           RETURN SymChar;
                        }

#charset# "}"           : {yyStart (STD); RETURN SymRBrace;}

< "#@" " " * "line" " " +:{yyPush (PPline);}

#PPline# {0-9} +        :- {GetWord (Word);
                            PPLine := StringToInt (Word) - 1;
                            (* -1 to compensate for the following yyEol () *)
                        }

#PPline# \" -{\n"} + \" :- {GetWord (Word);
                            SubString (Word, 2, Length (Word) - 1, Word);
                            yyLineCount := PPLine;
```

```
                        (* change the line only if there is a file name *)
                        Attribute.Position.File := MakeIdent (Word);
                    }

#PPline# " " * \r ? \n  :- {yyPop ();
                        CASE yyStartState OF
                        | targetcode:
                          AppendLine (TargetCode, yyLineCount,
                                      Attribute.Position.File);
                        ELSE
                        END;
                        yyEol (0); (* don't move yyEol before CASE *)
                    }
```

References

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

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

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

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

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

Contents