Selected Examples of Scanner Specifications

J. Grosch

GESELLSCHAFT FÜR MATHEMATIK
UND DATENVERARBEITUNG MBH

FORSCHUNGSSTELLE FÜR
PROGRAMMSTRUKTUREN
AN DER UNIVERSITÄT KARLSRUHE

Project

Compiler Generation

Selected Examples of Scanner Specifications

Josef Grosch

Mar. 8, 1988

Report No. 7

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

1. Introduction

Among the tokens to be recognized by scanners are a few that require non trivial processing: comments, strings, and character constants. Even identifiers and keywords may cause some trouble if the language defines upper-case and lower-case letters to have the same meaning. The problems with these tokens are the following:

- maintaining the line count during tokens extending on several lines
- maintaining the column count during tokens containing tab characters
- computation of the source position of tokens extending on several lines or of compound tokens which are recognized as a sequence of subtokens
- nested comments
- report unclosed strings and comments as errors
- computing the internal representation of strings
- conversion of escape sequences such as doubled string delimiters or preceding escape characters
- normalization of upper-case and lower-case letters

The following chapters contain solutions to the above problems for the languages Pascal, Modula, C, and Ada. The solutions are scanner specifications suitable as input for the scanner generator Rex [Gro87]. The primary intention of this paper is to serve as a reference manual containing examples for non trivial cases. All specifications use C as target language except the chapter on Modula which uses Modula. The Appendix contains a complete scanner specification for Ada with Modula as target language.

2. Pascal

2.1. Comments

Problems to solve:

- unclosed comments
- newline characters
- tab characters

Solution:

```
EOF      {IF yyStartState = Comment THEN Error ("unclosed comment"); END;}

DEFINE   CmtCh    = - {*\}\t\n}.

START    Comment

RULE

            "(*" | "{"      :- {yyStart (Comment);}
#Comment#   "*)" | "}"      :- {yyStart (STD);}
#Comment#   "*"  | CmtCh +  :- {}
```

Comments are processed in a separate start state called *Comment.* Everything is skipped in this state except closing comment brackets which switch back to start state STD. The single characters '*' or '}' which can start a closing comment bracket have to be skipped separately. Otherwise closing comment brackets would not be recognized because of the "longest match" rule of Rex. An unclosed comment is indicated by reaching end of file while in start state *Comment.* We presuppose the existence of a procedure *Error* to report this condition. We don't need to care about tab and newline characters other than excluding them from the set *CmtCh* because the two rules needed for this problem are already predefined by Rex:

```
#Comment#  \t :- {yyTab;}
#Comment#  \n :- {yyEol (0);}
```

2.2. Identifiers

Problems to solve:

- normalization of upper-case and lower-case letters

Solution:

```
EXPORT  {
# include "Idents.h"
# include "Positions.h"
typedef struct {
   tPosition Position;
   tIdent    Ident;
} tScanAttribute;
extern void ErrorAttribute ();
}

GLOBAL  {
# define TokIdentifier ...
void ErrorAttribute (Token, Attribute)
   int Token;
   tScanAttribute * Attribute;
   {
      Attribute->Ident = NoIdent;
   }
}

LOCAL   {char String [256]; int L;}

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

RULE

letter (letter | digit) * : {L = GetLower (String);
                Attribute.Ident = MakeIdent (String, L); return TokIdentifier;}
```

Normalization of upper-case and lower-case letters to lower-case is done by the predefined operation *GetLower* of Rex.

2.3. Character Constants

Problems to solve:

- conversion
- tab characters

Solution:

```
EXPORT  {
# include "Positions.h"
typedef struct {
   tPosition Position;
   char      Char;
} tScanAttribute;
extern void ErrorAttribute ();
}

GLOBAL  {
# define TokCharConst ...
void ErrorAttribute (Token, Attribute)
   int Token;
   tScanAttribute * Attribute;
   {
      Attribute->Char = '\0';
   }
}

RULE

''''     : {Attribute.Char = '\''; return TokCharConst;}
' \t '   : {Attribute.Char = '\t'; yyTab2 (1, 1); return TokCharConst;}
' ANY '  : {Attribute.Char = TokenPtr [1]; return TokCharConst;}
```

In this example the order of the rules is significant because the last rule would also match the characters of the preceding one.

2.4. Strings

Problems to solve:

- conversion
- doubled delimiters
- tab characters
- unclosed strings (at end of lines)
- source position

Solution:

```
EXPORT  {
# include "StringMem.h"
# include "Positions.h"
typedef struct {
   tPosition  Position;
   tStringRef StringRef;
} tScanAttribute;
extern void ErrorAttribute ();
}

GLOBAL  {
# define TokString ...
void ErrorAttribute (Token, Attribute) ...
}

LOCAL   {char String [256]; int L;}

DEFINE  StrCh   = - {'\t\n}.

START   string

RULE

#STD#    '      :  {yyStart (string); L = 0;}
#string# StrCh +:- {L += GetWord (& String [L]);}
#string# ''     :- {String [L ++] = '\'';}
#string# '      :- {yyStart (STD); String [L] = '\0';
                    Attribute.StringRef = PutString (String, L);
                    return TokString;}
#string# \t     :- {String [L ++] = '\t'; yyTab;}
#string# \n     :- {Error ("unclosed string"); yyEol (0);
                    yyStart (STD); String [L] = '\0';
                    Attribute.StringRef = PutString (String, L);
                    return TokString;}
```

We presuppose the existence of a string memory module *StringMem.* The procedure *PutString* stores a string in the string memory and returns a reference to it which can be used as attribute of the token *TokString*.

2.5. Keywords

Problems to solve:

- normalization of upper-case and lower-case letters

Solution:

```
GLOBAL  {
# define TokAND       ...
   ...
# define TokWITH         ...
void ErrorAttribute (Token, Attribute) ...
}

DEFINE  A = {Aa}.
          ...
        Z = {Zz}.

RULE

A N D           : {return TokAND       ;}
  ...
W I T H         : {return TokWITH      ;}
```

The idea of the solution is to define identifiers A to Z to stand for the corresponding upper-case as well as lower-case letters. Then specifying the keywords in upper-case and spaced does the job.

3. Modula

3.1. Comments

Problems to solve:

- nested comments
- unclosed comments
- newline characters
- tab characters

Solution:

```
GLOBAL  {VAR NestingLevel: CARDINAL;}

BEGIN   {NestingLevel := 0;}

EOF     {IF yyStartState = Comment THEN Error ("unclosed comment"); END;}

DEFINE  CmtCh   = - {*(\t\n}.

START   Comment

RULE

#STD, Comment# "(*" :- {INC (NestingLevel); yyStart (Comment);}
#Comment#  "*)" :- {DEC (NestingLevel);
                    IF NestingLevel = 0 THEN yyStart (STD); END;}
#Comment#  "(" | "*" | CmtCh + :- {}
```

We need a variable *NestingLevel* to count the nesting depth of comments because it is not possible to specify nested comments by a regular expression. Comments are processed in a separate start state called *Comment.* Everything is skipped in this state except opening or closing comment brackets which trigger a change of the nesting level. The single characters '(' and '*' which can start opening or closing comment brackets have to be skipped separately. Otherwise comment brackets within comment would not be recognized because of the "longest match" rule of Rex. An unclosed comment is indicated by reaching end of file while in start state *Comment.* We presuppose the existence of a procedure *Error* to report this condition. We don't need to care about tab and newline characters other than excluding them from the set *CmtCh* because the two rules needed for this problem are already predefined by Rex:

```
#Comment#  \t :- {yyTab;}
#Comment#  \n :- {yyEol (0);}
```

3.2. Strings

Problems to solve:

- conversion
- tab characters
- unclosed strings (at end of lines)
- source position

Solution:

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

GLOBAL  {
   FROM Strings     IMPORT tString, AssignEmpty, Concatenate, Append;
   FROM StringMem   IMPORT PutString;

   CONST TokString = ...;

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

LOCAL   {VAR String, S: tString;}

DEFINE  StrCh1  = - {'\t\n}.
        StrCh2  = - {"\t\n}.

START   Str1, Str2

RULE

#STD#   '       :  {AssignEmpty (String); yyStart (Str1);}
#Str1#  StrCh1+ :- {GetWord (S); Concatenate (String, S);}
#Str1#  '       :- {yyStart (STD);
                    Attribute.StringRef := PutString (String);
                    RETURN TokString;}

#STD#   \"      :  {AssignEmpty (String); yyStart (Str2);}
#Str2#  StrCh2+ :- {GetWord (S); Concatenate (String, S);}
#Str2#  \"      :- {yyStart (STD);
                    Attribute.StringRef := PutString (String);
                    RETURN TokString;}

#Str1, Str2# \t :- {Append (String, 11C); yyTab;}
#Str1, Str2# \n :- {Error ("unclosed string"); yyEol (0); yyStart (STD);
                    Attribute.StringRef := PutString (String);
                    RETURN TokString;}
```

Again two separate start states are used to recognize the two forms of Modula-2 strings. We presuppose the existence of a string handling module *Strings* and a string memory module *StringMem.* The procedure *PutString* stores a string in the string memory and returns a reference to it which can be used as attribute of the token *TokString.*

4. C

4.1. Comments

Problems to solve:

- unclosed comments
- newline characters
- tab characters

Solution:

```
EOF     {if (yyStartState == Comment) Error ("unclosed comment");}

DEFINE  CmtCh   = - {*\t\n}.

START   Comment

RULE

             "/*" :- {yyStart (Comment);}
#Comment#    "*/" :- {yyStart (STD);}
#Comment#    "*" | CmtCh + :- {}
```

Comments are processed in a separate start state called *Comment.* Everything is skipped in this state except closing comment brackets which switch back to start state STD. The single character '*' which can start a closing comment bracket has to be skipped separately. Otherwise closing comment brackets would not be recognized because of the "longest match" rule of Rex. An unclosed comment is indicated by reaching end of file while in start state *Comment.* We presuppose the existence of a procedure *Error* to report this condition. We don't need to care about tab and newline characters other than excluding them from the set *CmtCh* because the two rules needed for this problem are already predefined by Rex:

```
#Comment#  \t :- {yyTab;}
#Comment#  \n :- {yyEol (0);}
```

4.2. Character Constants

Problems to solve:

- conversion
- escape sequences
- tab characters

Solution:

```
EXPORT  {
# include "Positions.h"
typedef struct {
   tPosition Position;
   char      Char;
} tScanAttribute;
extern void ErrorAttribute ();
}

GLOBAL  {
# define TokChar ...
void ErrorAttribute (Token, Attribute)
   int Token;
   tScanAttribute * Attribute;
   {
      Attribute->Char = '\0';
   }
}

LOCAL   {char String [256];}

RULE

' \t '         : {Attribute.Char = '\t'; yyTab2 (1, 1); return TokChar;}
' ANY '        : {Attribute.Char = TokenPtr [1]; return TokChar;}
' \\ n '       : {Attribute.Char = '\n'; return TokChar;}
' \\ t '       : {Attribute.Char = '\t'; return TokChar;}
' \\ v '       : {Attribute.Char = '\v'; return TokChar;}
' \\ b '       : {Attribute.Char = '\b'; return TokChar;}
' \\ r '       : {Attribute.Char = '\r'; return TokChar;}
' \\ f '       : {Attribute.Char = '\f'; return TokChar;}
' \\ {0-7}[1-3] ' : {(void) GetWord (String);
                 sscanf (String + 2, "%o", & Attribute.Char);
                 return TokChar;}
' \\ ANY '     : {Attribute.Char = TokenPtr [2]; return TokChar;}
```

In this example the order of the rules is significant because the second rule would also match the characters of the first one. The same holds for the group of following rules with respect to the last rule.

4.3. Strings

Problems to solve:

- conversion
- escape sequences
- tab characters
- strings ranging over several lines
- source position

Solution:

```
EXPORT  {
# include "StringMem.h"
# include "Positions.h"
typedef struct {
   tPosition  Position;
   tStringRef StringRef;
} tScanAttribute;
extern void ErrorAttribute ();
}

GLOBAL  {
# define TokString ...
void ErrorAttribute (Token, Attribute) ...
}

LOCAL   {char String [256], S [5]; int L;}

DEFINE  StrCh   = - {"\t\n\\}.

START   string

RULE

#STD#    \"     :  {yyStart (string); L = 0;}
#string# StrCh+ :- {L += GetWord (& String [L]);}
#string# \t     :- {String [L ++] = '\t'); yyTab;}
#string# \\ n   :- {String [L ++] = '\n');}
#string# \\ t   :- {String [L ++] = '\t');}
#string# \\ v   :- {String [L ++] = '\v');}
#string# \\ b   :- {String [L ++] = '\b');}
#string# \\ r   :- {String [L ++] = '\r');}
#string# \\ f   :- {String [L ++] = '\f');}
#string# \\ {0-7}[1-3] :- {(void) GetWord (S);
                    sscanf (S + 1, "%o", & String [L ++]);}
#string# \\ ANY :- {(void) GetWord (S); String [L ++] = S [1];}
#string# \\ \n  :- {yyEol (0); String [L ++] = '\n';}
#string# \"     :- {yyStart (STD); String [L] = '\0';
                    Attribute.StringRef = PutString (String, L);
                    return TokString;}
#string# \n     :- {Error ("unclosed string"); yyEol (0);
                    yyStart (STD); String [L] = '\0';
                    Attribute.StringRef = PutString (String, L);
                    return TokString;}
```

We presuppose the existence of a string memory module *StringMem.* The procedure *PutString* stores a string in the string memory and returns a reference to it which can be used as attribute of the token *TokString.*

5. Ada

5.1. Identifiers

Problems to solve:

- normalization of upper-case and lower-case letters

Solution:

```
EXPORT  {
# include "Idents.h"
# include "Positions.h"
typedef struct {
   tPosition Position;
   tIdent    Ident;
} tScanAttribute;
extern void ErrorAttribute ();
}

GLOBAL  {
# define TokIdentifier ...
void ErrorAttribute (Token, Attribute) ...
}

LOCAL   {char String [256]; int L;}

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

RULE

letter (_? (letter | digit)+ )* : {L = GetLower (String);
                Attribute.Ident = MakeIdent (String, L); return TokIdentifier;}
```

Normalization of upper-case and lower-case letters to lower-case is done by the predefined operation *GetLower* of Rex.

5.2. Numeric Literals

Problems to solve:

- conversion

Solution:

```
EXPORT  {
# include "StringMem.h"
# include "Positions.h"
typedef struct {
   tPosition  Position;
   tStringRef StringRef;
} tScanAttribute;
extern void ErrorAttribute ();
}

GLOBAL  {
# define TokDecimalLiteral ...
# define TokBasedLiteral ...
void ErrorAttribute (Token, Attribute) ...
}

DEFINE  digit           = {0-9}                 .
        extended_digit  = digit | {A-F a-f}     .
        integer         = digit (_? digit) *    .
        based_integer   = extended_digit (_? extended_digit) *  .
        exponent        = {Ee} {+\-} ? integer .

RULE

integer ("." integer) ? exponent ? :
                 {Attribute.StringRef = PutString (TokenPtr, TokenLength);
                  return TokDecimalLiteral;}

integer "#" based_integer ("." based_integer) ? "#" exponent ? :
                 {Attribute.StringRef = PutString (TokenPtr, TokenLength);
                  return TokBasedLiteral;}
```

The conversion of numeric literals to numeric values is not really solved in the above solution. By storing the external representation of numeric literals in a string memory the values are treated symbolically and true conversion is delayed to be done by other compiler phases.

5.3. Character Literals

Problems to solve:

- no problems to solve for character literals
- distinction between character literals and apostrophes

Solution:

```
EXPORT  {
# include "Idents.h"
# include "Positions.h"
typedef struct {
   tPosition Position;
   char      Char;
   tIdent    Ident;
} tScanAttribute;
extern void ErrorAttribute ();
}

GLOBAL  {
# define TokIdentifier          ...
# define TokCharacterLiteral    ...
# define TokApostrophe          ...
# define TokLParenthesis        ...
# define TokRParenthesis        ...
void ErrorAttribute (Token, Attribute) ...
}

LOCAL   {char String [256]; int L;}

DEFINE  character = {\ -~}.
        letter    = {A-Z a-z}.
        digit     = {0-9}.

START   QUOTE

RULE

#STD#   ' character '    : {Attribute.Char = TokenPtr [1];
                            return TokCharacterLiteral;}
#QUOTE# '                : {yyStart (STD); return TokApostrophe;}
        "("              : {yyStart (STD); return TokLParenthesis;}
        ")"              : {yyStart (QUOTE); return TokRParenthesis;}
        letter (_? (letter | digit)+ )*
                         : {yyStart (QUOTE); L = GetLower (Word);
                            Attribute.Ident = MakeIdent (Word, L);
                            return TokIdentifier;}
```

The tokens *Character Literal* and *Apostrophe* can be distinguished in Ada only by consideration of some context. The pathological input is for example something like

```
t'('a','b','c')
```

where t is a type_mark used as qualification for an aggregate of character literals. It has to be taken care that 'a', 'b', and 'c' are recognized as character literals and not '(', ',', and ','. Studying the Ada grammar one can see that apostrophes are used following identifiers and closing parentheses only. There are never character literals in this places.

This leads to the above solution with an additional start state called *QUOTE*. After recognition of an identifier or a closing parentheses the scanner is switched to start state *QUOTE*. After recognition of all other tokens the scanner is switched back to start state *STD*.

Apostrophes are recognized only in start state *QUOTE* and character literals only in start state *STD.* All the other tokens are recognized in both start states.

5.4. String Literals

Problems to solve:

- conversion
- doubled delimiters
- unclosed strings (at end of lines)
- source position

Solution:

```
EXPORT  {
# include "StringMem.h"
# include "Positions.h"
typedef struct {
   tPosition  Position;
   tStringRef StringRef;
} tScanAttribute;
extern void ErrorAttribute ();
}

GLOBAL  {
# define TokStringLiteral ...
void ErrorAttribute (Token, Attribute) ...
}

LOCAL   {char String [256]; int L;}

DEFINE  StrCh   = {\ !#-~}.

START   string

RULE

#STD#    \"     :  {yyStart (string); L = 0;}
#string# StrCh+ :- {L += GetWord (& String [L]);}
#string# \"\"   :- {String [L ++] = '\"';}
#string# \"     :- {yyStart (STD); String [L] = '\0';
                    Attribute.StringRef = PutString (String, L);
                    return TokStringLiteral;}
#string# \n     :- {Error ("unclosed string"); yyEol (0);
                    yyStart (STD); String [L] = '\0';
                    Attribute.StringRef = PutString (String, L);
                    return TokStringLiteral;}
```

We presuppose the existence of a string memory module *StringMem.* The procedure *PutString* stores a string in the string memory and returns a reference to it which can be used as attribute of the token *TokString.*

5.5. Keywords

Problems to solve:

- normalization of upper-case and lower-case letters

Solution:

```
GLOBAL  {
# define TokABORT       ...
   ...
# define TokXOR         ...
void ErrorAttribute (Token, Attribute) ...
}

DEFINE  A = {Aa}.
          ...
        Z = {Zz}.

RULE

A B O R T       : {return TokABORT     ;}
  ...
X O R           : {return TokXOR       ;}
```

The idea of the solution is to define identifiers A to Z to stand for the corresponding upper-case as well as lower-case letters. Then specifying the keywords in upper-case and spaced does the job.

Appendix: Complete Scanner Specification for Ada

```
GLOBAL  {
   FROM Strings          IMPORT tString, AssignEmpty, Concatenate, Append, Char;
   FROM StringMem        IMPORT tStringRef, PutString;
   FROM Idents           IMPORT tIdent, MakeIdent;

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

CONST
   TokIdentifier         =  1    ;
   TokDecimalLiteral     =  2    ;
   TokBasedLiteral       =  3    ;
   TokCharLiteral        =  4    ;
   TokStringLiteral      =  5    ;

   TokArrow              =  6    ;        (* '=>'           *)
   TokDoubleDot          =  7    ;        (* '..'           *)
   TokDoubleStar         =  8    ;        (* '**'           *)
   TokBecomes            =  9    ;        (* ':='           *)
   TokNotEqual           = 10    ;        (* '/='           *)
   TokGreaterEqual       = 11    ;        (* '>='           *)
   TokLessEqual          = 12    ;        (* '<='           *)
   TokLLabelBracket      = 13    ;        (* '<<'           *)
   TokRLabelBracket      = 14    ;        (* '>>'           *)
   TokBox                = 15    ;        (* '<>'           *)

   TokAmpersand          = 16    ;        (* '&'            *)
   TokApostrophe         = 17    ;        (* '''            *)
   TokLParenthesis       = 18    ;        (* '('            *)
   TokRParenthesis       = 19    ;        (* ')'            *)
   TokStar               = 20    ;        (* '*'            *)
   TokPlus               = 21    ;        (* '+'            *)
   TokComma              = 22    ;        (* ','            *)
   TokMinus              = 23    ;        (* '-'            *)
   TokDot                = 24    ;        (* '.'            *)
   TokDivide             = 25    ;        (* '/'            *)
   TokColon              = 26    ;        (* ':'            *)
   TokSemicolon          = 27    ;        (* ';'            *)
   TokLess               = 28    ;        (* '<'            *)
   TokEqual              = 29    ;        (* '='            *)
   TokGreater            = 30    ;        (* '>'            *)
   TokBar                = 31    ;        (* '|'            *)

   TokABORT              = 32    ;        (* ABORT          *)
   TokABS                = 33    ;        (* ABS            *)
   TokACCEPT             = 34    ;        (* ACCEPT         *)
   TokACCESS             = 35    ;        (* ACCESS         *)
   TokALL                = 36    ;        (* ALL            *)
   TokAND                = 37    ;        (* AND            *)
   TokARRAY              = 38    ;        (* ARRAY          *)
   TokAT                 = 39    ;        (* AT             *)
   TokBEGIN              = 40    ;        (* BEGIN          *)
   TokBODY               = 41    ;        (* BODY           *)
   TokCASE               = 42    ;        (* CASE           *)
   TokCONSTANT           = 43    ;        (* CONSTANT       *)
   TokDECLARE            = 44    ;        (* DECLARE        *)
   TokDELAY              = 45    ;        (* DELAY          *)
   TokDELTA              = 46    ;        (* DELTA          *)
```

```
   TokDIGITS            = 47    ;       (* DIGITS       *)
   TokDO                = 48    ;       (* DO           *)
   TokELSE              = 49    ;       (* ELSE         *)
   TokELSIF             = 50    ;       (* ELSIF        *)
   TokEND               = 51    ;       (* END          *)
   TokENTRY             = 52    ;       (* ENTRY        *)
   TokEXCEPTION         = 53    ;       (* EXCEPTION    *)
   TokEXIT              = 54    ;       (* EXIT         *)
   TokFOR               = 55    ;       (* FOR          *)
   TokFUNCTION          = 56    ;       (* FUNCTION     *)
   TokGENERIC           = 57    ;       (* GENERIC      *)
   TokGOTO              = 58    ;       (* GOTO         *)
   TokIF                = 59    ;       (* IF           *)
   TokIN                = 60    ;       (* IN           *)
   TokIS                = 61    ;       (* IS           *)
   TokLIMITED           = 62    ;       (* LIMITED      *)
   TokLOOP              = 63    ;       (* LOOP         *)
   TokMOD               = 64    ;       (* MOD          *)
   TokNEW               = 65    ;       (* NEW          *)
   TokNOT               = 66    ;       (* NOT          *)
   TokNULL              = 67    ;       (* NULL         *)
   TokOF                = 68    ;       (* OF           *)
   TokOR                = 69    ;       (* OR           *)
   TokOTHERS            = 70    ;       (* OTHERS       *)
   TokOUT               = 71    ;       (* OUT          *)
   TokPACKAGE           = 72    ;       (* PACKAGE      *)
   TokPRAGMA            = 73    ;       (* PRAGMA       *)
   TokPRIVATE           = 74    ;       (* PRIVATE      *)
   TokPROCEDURE         = 75    ;       (* PROCEDURE    *)
   TokRAISE             = 76    ;       (* RAISE        *)
   TokRANGE             = 77    ;       (* RANGE        *)
   TokRECORD            = 78    ;       (* RECORD       *)
   TokREM               = 79    ;       (* REM          *)
   TokRENAMES           = 80    ;       (* RENAMES      *)
   TokRETURN            = 81    ;       (* RETURN       *)
   TokREVERSE           = 82    ;       (* REVERSE      *)
   TokSELECT            = 83    ;       (* SELECT       *)
   TokSEPARATE          = 84    ;       (* SEPARATE     *)
   TokSUBTYPE           = 85    ;       (* SUBTYPE      *)
   TokTASK              = 86    ;       (* TASK         *)
   TokTERMINATE         = 87    ;       (* TERMINATE    *)
   TokTHEN              = 88    ;       (* THEN         *)
   TokTYPE              = 89    ;       (* TYPE         *)
   TokUSE               = 90    ;       (* USE          *)
   TokWHEN              = 91    ;       (* WHEN         *)
   TokWHILE             = 92    ;       (* WHILE        *)
   TokWITH              = 93    ;       (* WITH         *)
   TokXOR               = 94    ;       (* XOR          *)
}

LOCAL   {
   VAR
      String, S : tString       ;
      Word      : tString       ;
      ident     : tIdent        ;
      string    : tStringRef    ;
      ch        : CHAR          ;
}

DEFINE
   digit          = {0-9}               .
```

```
   extended_digit = digit | {A-F a-f}   .
   letter         = {a-z A-Z}           .
   character      = {\ -~}              .
   stringch       = {\ !#-~}            .
   integer        = digit (_? digit) *  .
   based_integer  = extended_digit (_? extended_digit) *  .
   illegal        = - {\ \t\n}          .

   A            = {Aa}  .
   B            = {Bb}  .
   C            = {Cc}  .
   D            = {Dd}  .
   E            = {Ee}  .
   F            = {Ff}  .
   G            = {Gg}  .
   H            = {Hh}  .
   I            = {Ii}  .
   J            = {Jj}  .
   K            = {Kk}  .
   L            = {Ll}  .
   M            = {Mm}  .
   N            = {Nn}  .
   O            = {Oo}  .
   P            = {Pp}  .
   Q            = {Qq}  .
   R            = {Rr}  .
   S            = {Ss}  .
   T            = {Tt}  .
   U            = {Uu}  .
   V            = {Vv}  .
   W            = {Ww}  .
   X            = {Xx}  .
   Y            = {Yy}  .
   Z            = {Zz}  .

START   STRING, QUOTE

RULE

NOT #STRING# integer ("." integer) ? (E {+\-} ? integer) ?
                        : {yyStart (STD); GetWord (Word);
                           string := PutString (Word);
                           RETURN TokDecimalLiteral;}

NOT #STRING#
    integer "#" based_integer ("." based_integer) ? "#" (E {+\-} ? integer) ?
                        : {yyStart (STD); GetWord (Word);
                           string := PutString (Word);
                           RETURN TokBasedLiteral;}

    #STD#  ' character ': {GetWord (String); ch := Char (String, 2);
                           RETURN TokCharLiteral;}

NOT #STRING# \"         :  {yyStart (STRING); AssignEmpty (String);}
    #STRING# stringch + :- {GetWord (S); Concatenate (String, S);}
    #STRING# \"\"       :- {Append (String, '"');}
    #STRING# \"         :- {yyStart (STD); string := PutString (String);
                            RETURN TokStringLiteral;}
    #STRING# \t         :- {Append (String, 11C); yyTab;}
    #STRING# \n         :- {(* Error ("unclosed string"); *) yyEol (0);
                            yyStart (STD); string := PutString (String);
```

```
                                RETURN TokStringLiteral;}

NOT #STRING# "--" ANY * : {}

NOT #STRING# "=>"        : {yyStart (STD); RETURN TokArrow          ;}
NOT #STRING# ".."        : {yyStart (STD); RETURN TokDoubleDot      ;}
NOT #STRING# "**"        : {yyStart (STD); RETURN TokDoubleStar     ;}
NOT #STRING# ":="        : {yyStart (STD); RETURN TokBecomes        ;}
NOT #STRING# "/="        : {yyStart (STD); RETURN TokNotEqual       ;}
NOT #STRING# ">="        : {yyStart (STD); RETURN TokGreaterEqual   ;}
NOT #STRING# "<="        : {yyStart (STD); RETURN TokLessEqual      ;}
NOT #STRING# "<<"        : {yyStart (STD); RETURN TokLLabelBracket  ;}
NOT #STRING# ">>"        : {yyStart (STD); RETURN TokRLabelBracket  ;}
NOT #STRING# "<>"        : {yyStart (STD); RETURN TokBox            ;}

NOT #STRING# "&"         : {yyStart (STD); RETURN TokAmpersand      ;}
    #QUOTE#  "'"         : {yyStart (STD); RETURN TokApostrophe     ;}
NOT #STRING# "("         : {yyStart (STD); RETURN TokLParenthesis   ;}
NOT #STRING# ")"         : {yyStart (QUOTE); RETURN TokRParenthesis ;}
NOT #STRING# "*"         : {yyStart (STD); RETURN TokStar           ;}
NOT #STRING# "+"         : {yyStart (STD); RETURN TokPlus           ;}
NOT #STRING# ","         : {yyStart (STD); RETURN TokComma          ;}
NOT #STRING# "-"         : {yyStart (STD); RETURN TokMinus          ;}
NOT #STRING# "."         : {yyStart (STD); RETURN TokDot            ;}
NOT #STRING# "/"         : {yyStart (STD); RETURN TokDivide         ;}
NOT #STRING# ":"         : {yyStart (STD); RETURN TokColon          ;}
NOT #STRING# ";"         : {yyStart (STD); RETURN TokSemicolon      ;}
NOT #STRING# "<"         : {yyStart (STD); RETURN TokLess           ;}
NOT #STRING# "="         : {yyStart (STD); RETURN TokEqual          ;}
NOT #STRING# ">"         : {yyStart (STD); RETURN TokGreater        ;}
NOT #STRING# "|"         : {yyStart (STD); RETURN TokBar            ;}

NOT #STRING# A B O R T           : {yyStart (STD); RETURN TokABORT     ;}
NOT #STRING# A B S               : {yyStart (STD); RETURN TokABS       ;}
NOT #STRING# A C C E P T         : {yyStart (STD); RETURN TokACCEPT    ;}
NOT #STRING# A C C E S S         : {yyStart (STD); RETURN TokACCESS    ;}
NOT #STRING# A L L               : {yyStart (STD); RETURN TokALL       ;}
NOT #STRING# A N D               : {yyStart (STD); RETURN TokAND       ;}
NOT #STRING# A R R A Y           : {yyStart (STD); RETURN TokARRAY     ;}
NOT #STRING# A T                 : {yyStart (STD); RETURN TokAT        ;}
NOT #STRING# B E G I N           : {yyStart (STD); RETURN TokBEGIN     ;}
NOT #STRING# B O D Y             : {yyStart (STD); RETURN TokBODY      ;}
NOT #STRING# C A S E             : {yyStart (STD); RETURN TokCASE      ;}
NOT #STRING# C O N S T A N T     : {yyStart (STD); RETURN TokCONSTANT  ;}
NOT #STRING# D E C L A R E       : {yyStart (STD); RETURN TokDECLARE   ;}
NOT #STRING# D E L A Y           : {yyStart (STD); RETURN TokDELAY     ;}
NOT #STRING# D E L T A           : {yyStart (STD); RETURN TokDELTA     ;}
NOT #STRING# D I G I T S         : {yyStart (STD); RETURN TokDIGITS    ;}
NOT #STRING# D O                 : {yyStart (STD); RETURN TokDO        ;}
NOT #STRING# E L S E             : {yyStart (STD); RETURN TokELSE      ;}
NOT #STRING# E L S I F           : {yyStart (STD); RETURN TokELSIF     ;}
NOT #STRING# E N D               : {yyStart (STD); RETURN TokEND       ;}
NOT #STRING# E N T R Y           : {yyStart (STD); RETURN TokENTRY     ;}
NOT #STRING# E X C E P T I O N   : {yyStart (STD); RETURN TokEXCEPTION ;}
NOT #STRING# E X I T             : {yyStart (STD); RETURN TokEXIT      ;}
NOT #STRING# F O R               : {yyStart (STD); RETURN TokFOR       ;}
NOT #STRING# F U N C T I O N     : {yyStart (STD); RETURN TokFUNCTION  ;}
NOT #STRING# G E N E R I C       : {yyStart (STD); RETURN TokGENERIC   ;}
NOT #STRING# G O T O             : {yyStart (STD); RETURN TokGOTO      ;}
NOT #STRING# I F                 : {yyStart (STD); RETURN TokIF        ;}
NOT #STRING# I N                 : {yyStart (STD); RETURN TokIN        ;}
```

```
NOT #STRING# I S                  : {yyStart (STD); RETURN TokIS          ;}
NOT #STRING# L I M I T E D        : {yyStart (STD); RETURN TokLIMITED     ;}
NOT #STRING# L O O P              : {yyStart (STD); RETURN TokLOOP        ;}
NOT #STRING# M O D                : {yyStart (STD); RETURN TokMOD         ;}
NOT #STRING# N E W                : {yyStart (STD); RETURN TokNEW         ;}
NOT #STRING# N O T                : {yyStart (STD); RETURN TokNOT         ;}
NOT #STRING# N U L L              : {yyStart (STD); RETURN TokNULL        ;}
NOT #STRING# O F                  : {yyStart (STD); RETURN TokOF          ;}
NOT #STRING# O R                  : {yyStart (STD); RETURN TokOR          ;}
NOT #STRING# O T H E R S          : {yyStart (STD); RETURN TokOTHERS      ;}
NOT #STRING# O U T                : {yyStart (STD); RETURN TokOUT         ;}
NOT #STRING# P A C K A G E        : {yyStart (STD); RETURN TokPACKAGE     ;}
NOT #STRING# P R A G M A          : {yyStart (STD); RETURN TokPRAGMA      ;}
NOT #STRING# P R I V A T E        : {yyStart (STD); RETURN TokPRIVATE     ;}
NOT #STRING# P R O C E D U R E    : {yyStart (STD); RETURN TokPROCEDURE   ;}
NOT #STRING# R A I S E            : {yyStart (STD); RETURN TokRAISE       ;}
NOT #STRING# R A N G E            : {yyStart (STD); RETURN TokRANGE       ;}
NOT #STRING# R E C O R D          : {yyStart (STD); RETURN TokRECORD      ;}
NOT #STRING# R E M                : {yyStart (STD); RETURN TokREM         ;}
NOT #STRING# R E N A M E S        : {yyStart (STD); RETURN TokRENAMES     ;}
NOT #STRING# R E T U R N          : {yyStart (STD); RETURN TokRETURN      ;}
NOT #STRING# R E V E R S E        : {yyStart (STD); RETURN TokREVERSE     ;}
NOT #STRING# S E L E C T          : {yyStart (STD); RETURN TokSELECT      ;}
NOT #STRING# S E P A R A T E      : {yyStart (STD); RETURN TokSEPARATE    ;}
NOT #STRING# S U B T Y P E        : {yyStart (STD); RETURN TokSUBTYPE     ;}
NOT #STRING# T A S K              : {yyStart (STD); RETURN TokTASK        ;}
NOT #STRING# T E R M I N A T E    : {yyStart (STD); RETURN TokTERMINATE   ;}
NOT #STRING# T H E N              : {yyStart (STD); RETURN TokTHEN        ;}
NOT #STRING# T Y P E              : {yyStart (STD); RETURN TokTYPE        ;}
NOT #STRING# U S E                : {yyStart (STD); RETURN TokUSE         ;}
NOT #STRING# W H E N              : {yyStart (STD); RETURN TokWHEN        ;}
NOT #STRING# W H I L E            : {yyStart (STD); RETURN TokWHILE       ;}
NOT #STRING# W I T H              : {yyStart (STD); RETURN TokWITH        ;}
NOT #STRING# X O R                : {yyStart (STD); RETURN TokXOR         ;}

NOT #STRING# letter (_? (letter | digit)+ )*
                                  : {yyStart (QUOTE); GetLower (Word);
                                     ident := MakeIdent (Word);
                                     RETURN TokIdentifier;}

NOT #STRING# illegal              : {IO.WriteS (IO.StdOutput, "illegal character: ");
                                     yyEcho; IO.WriteNl (IO.StdOutput);}
```

References

[Gro87] J. Grosch, Rex - A Scanner Generator, Compiler Generation Report No. 5, GMD Forschungsstelle an der Universität Karlsruhe, Dec. 1987.

Contents