Syntaxe du langage à implémenter

PROGRAM ::= program ID ; BLOCK .
BLOCK ::= CONSTS VARS INSTS
CONSTS ::= const ID = NUM ; { ID = NUM ; } | e
VARS ::= var ID { , ID } ; | e
INSTS ::= begin INST { ; INST } end
INST ::= INSTS | AFFEC | SI | TANTQUE | ECRIRE | LIRE | e
AFFEC ::= ID := EXPR
SI ::= if COND then INST
TANTQUE ::= while COND do INST
ECRIRE ::= write (EXPR { , EXPR })
LIRE ::= read (ID { , ID })
COND ::= EXPR RELOP EXPR
RELOP ::= = | <> | < | > | <= | >=
EXPR ::= TERM { ADDOP TERM }
ADDOP ::= + | -
TERM ::= FACT { MULOP FACT }
MULOP ::= * | /
FACT ::= ID | NUM | (EXPR)

Analyseur lexical

Certains non-terminaux ne sont pas décrits par cette grammaire. Il s'agit des non-terminaux pris en charge par l'analyse lexicale :
ID représente les identificateurs, c'est-à-dire toute suite de lettres ou chiffres commençant par une lettre et qui ne représente pas un mot clé (qui sont les terminaux présents dans la grammaire) ;
NUM représente les constantes numériques, c'est-à-dire toute suite de chiffres.

L'analyseur lexical peut être vu comme une procédure appelée par l'analyseur syntaxique. Chaque appel à la procédure d'analyse lexical NEXT_TOKEN met à jour la variable TOKEN
TOKEN contient le dernier token lu ;

Il faut donc écrire la procédure analyse-lexicale qui teste si la syntaxe du langage est bien respectée.

Analyseur lexical

L'analyseur lexical peut être vu comme une procédure appelée par l'analyseur syntaxique. Chaque appel à la procédure d'analyse lexical NEXT_TOKEN met à jour les variables TOKEN, SYM, et VAL :

TOKEN contient le dernier token lu ;
SYM contient la forme textuelle du dernier token lu ;
VAL est la valeur du dernier token lu.

Analyseur lexical

```
type  TOKENS = (ID_TOKEN, NUM_TOKEN, PLUS_TOKEN, MOINS_TOKEN, MUL_TOKEN
        DIV_TOKEN, EGAL_TOKEN, DIFF_TOKEN, INF_TOKEN, SUP_TOKEN,
        INF_EGAL_TOKEN, SUP_EGAL_TOKEN, PAR_OUV_TOKEN,
        PAR_FER_TOKEN, VIRG_TOKEN, PT_VIRG_TOKEN, POINT_TOKEN,
        AFFEC_TOKEN, BEGIN_TOKEN, END_TOKEN, IF_TOKEN ,WHILE_TOKEN,
        THEN_TOKEN, DO_TOKEN, WRITE_TOKEN, READ_TOKEN,
        CONST_TOKEN, VAR_TOKEN, PROGRAM_TOKEN, TOKEN_INCONNU) ;
      ALFA = packed array [1 .. 8] of char ;
var   TOKEN : TOKENS ;
      SYM : ALFA ;
      VAL : integer ;
```

Analyseur lexical

Une procédure TESTE teste si le prochain token (TOKEN) est bien celui passé en paramètre à la procédure ; on s'arrête sur une erreur sinon (procédure ERREUR) :

```
procedure TESTE (T:TOKENS) ;
begin
   if TOKEN = T
      then NEXT_TOKEN
      else ERREUR
end ;
```

Analyseur lexical

L'idée est que chaque règle de la grammaire est associée à une procédure qui << vérifie >> la concordance du texte à analyser avec une de ses parties droites.

Voici la fonction principale d'un tel analyseur :

```
procedure PROGRAM;
begin
   TESTE(PROGRAM_TOKEN) ;
   TESTE (ID_TOKEN) ;
   TESTE (PT_VIRG_TOKEN) ;
   BLOCK ;
   if TOKEN <> POINT_TOKEN then ERREUR
end ;
```

Analyseur lexical

La procédure BLOCK est la suivante :

```
procedure BLOCK ;
begin
   if TOKEN = CONST_TOKEN then CONSTS ;
   if TOKEN = VAR_TOKEN then VARS ;
   INSTS
end
```

Analyseur lexical

Les autres procédures sont :

```
procedure CONSTS ;
begin
   TESTE (CONST_TOKEN) ;
   repeat
      TESTE (ID_TOKEN) ;
      TESTE (EGAL_TOKEN) ;
      TESTE (NUM_TOKEN) ;
      TESTE (PT_VIRG_TOKEN)
   until TOKEN <> ID_TOKEN
end ;
```

Analyseur lexical

```
procedure VARS ;
 begin
    TESTE (VAR_TOKEN) ;
    TESTE (ID_TOKEN) ;
    while TOKEN = VIRG_TOKEN do
      begin NEXT_TOKEN ; TESTE (ID_TOKEN) end ;
    TESTE (PT_VIRG_TOKEN)
 end ;
```

Analyseur lexical

```
procedure INSTS ;
begin
   TESTE (BEGIN_TOKEN) ;
   INST ;
   while TOKEN = PT_VIRG_TOKEN do
     begin NEXT_TOKEN ; INST end ;
   TESTE (END_TOKEN)
end ;
procedure INST ;
begin
   case TOKEN of
     ID_TOKEN : AFFEC ;
     IF_TOKEN : SI ;
     WHILE_TOKEN : TANTQUE ;
     BEGIN_TOKEN : INSTS ;
     WRITE_TOKEN : ECRIRE ;
     READ_TOKEN : LIRE
   end
end ;
```

Analyseur lexical

```
procedure AFFEC ;
begin
   TESTE (ID_TOKEN) ;
   TESTE (AFFEC_TOKEN) ;
   EXPR
end ;
procedure SI ;
begin
   TESTE (IF_TOKEN) ;
   COND ;
   TESTE (THEN_TOKEN) ;
   INST
end ;
```

Analyseur lexical

```
procedure TANTQUE ;
begin
   TESTE (WHILE_TOKEN) ;
   COND ;
   TESTE (DO_TOKEN) ;
   INST
end ;
procedure ECRIRE ;
begin
   TESTE (WRITE_TOKEN) ;
   TESTE (PAR_OUV_TOKEN) ;
   EXPR ;
   while TOKEN = VIRG_TOKEN do
     begin NEXT_TOKEN ; EXPR end ;
   TESTE (PAR_FER_TOKEN) ;
end ;
```

Analyseur lexical

```
procedure LIRE ;
begin
   TESTE (READ_TOKEN) ;
   TESTE (PAR_OUV_TOKEN) ;
   TESTE (ID_TOKEN) ;
   while TOKEN = VIRG_TOKEN do
     begin NEXT_TOKEN ; TESTE (ID_TOKEN) end ;
   TESTE (PAR_FER_TOKEN)
end ;
procedure EXPR ;
begin
   TERM ;
   while TOKEN in [PLUS_TOKEN, MOINS_TOKEN] do
     begin NEXT_TOKEN ; TERM end
end ;
```

Analyseur lexical

```
procedure COND ;
begin
  EXPR ;
  if TOKEN in [EGAL_TOKEN, DIFF_TOKEN, INF_TOKEN, SUP_TOKEN,
    INF_EGAL_TOKEN, SUP_EGAL_TOKEN ]
    then begin
      NEXT_TOKEN ;
      EXPR
    end
end ;
procedure TERM ;
begin
  FACT ;
  while TOKEN in [MULT_TOKEN, DIV_TOKEN] do
    begin NEXT_TOKEN ; FACT end
end ;
```

Analyseur lexical

```
procedure FACT ;
begin
   if TOKEN in [ID_TOKEN, NUM_TOKEN]
      then NEXT_TOKEN
   else begin
      TESTE (PAR_OUV_TOKEN) ;
      EXPR ;
      TESTE (PAR_FER_TOKEN)
   end
```

Cet analyseur s'arrête à la première erreur détectée. Nous verrons comment améliorer cette situation.