# Syntaxte 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 )
```

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.

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.

```
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;
```

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;
```

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;
```

La procédure BLOCK est la suivante :

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

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:

```
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;
```

```
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;
```

```
procedure AFFEC;
begin
 TESTE (ID TOKEN);
 TESTE (AFFEC TOKEN);
 EXPR
end;
procedure SI;
begin
 TESTE (IF TOKEN);
 COND;
 TESTE (THEN TOKEN);
 INST
end;
```

```
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;
```

```
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;
```

```
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;
```

```
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.