

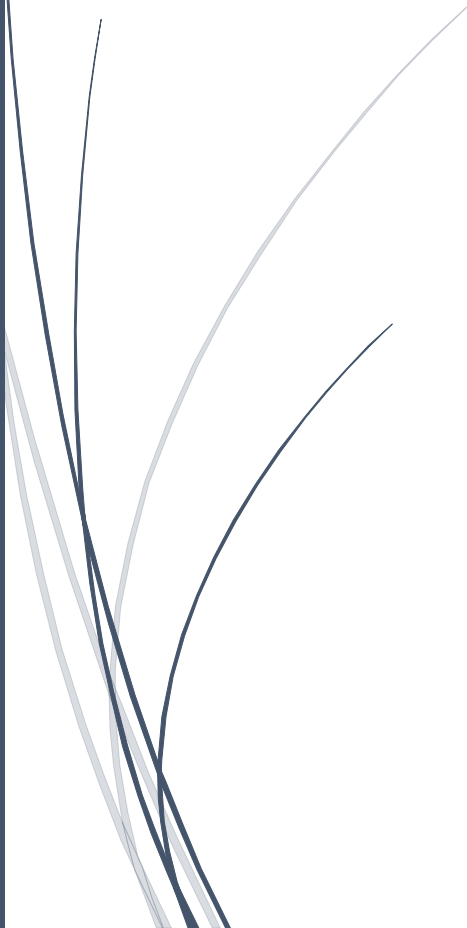
FAIT PAR : MOUAD RIALI — KAMAL ADDI

RAPPORT

>

Analyseur Syntaxique

Langage pascal



Analyseur Syntaxique :

Un analyseur syntaxique (parser) est basé sur un accepteur, fonction booléenne indiquant si le texte source est conforme aux règles de grammaire définissant le langage.

Forme du fichier Analyseur_SYN.c :

Définitions et Déclarations :

Où on peut inclure les bibliothèques C et déclarer les variables, les énumérations, les structures ...

```
Analyseur_SYN.c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <ctype.h>
5  #include <conio.h>
6
7  FILE* f;
8  char Car_Cour;
9  char *pchar,*pchar_temp, chaine[20],chainePR[20];
10
11  typedef enum{
12      //Mots cles :
13      PROGRAM_TOKEN,CONST_TOKEN,VAR_TOKEN,BEGIN_TOKEN,END_TOKEN,IF_TOKEN,THEN_TOKEN,
14      WHILE_TOKEN,DO_TOKEN,READ_TOKEN,WRITE_TOKEN,REPEAT_TOKEN,FOR_TOKEN,CASE_TOKEN,ELSE_TOKEN,
15      //LES SYMBOLES SPECIAUX :
16      DP_TOKEN,PV_TOKEN,PT_TOKEN,PLUS_TOKEN,MOINS_TOKEN,MULT_TOKEN,DIV_TOKEN,VIR_TOKEN,AFF_TOKEN,
17      INF_TOKEN,INFEG_TOKEN,SUP_TOKEN,SUEG_TOKEN,DIFF_TOKEN,PO_TOKEN,PF_TOKEN,FIN_TOKEN,EG_TOKEN,OF_TOKEN,UNTIL_TOKEN,
18      //Autres :
19      INTO_TOKEN,
20      DOWNTOKEN,
21      ID_TOKEN,
22      NUM_TOKEN,
23      ERROR_TOKEN,
24      COM_TOKEN,ACC_TOKEN
25  }CODE_LEX;
26
```

Et aussi des signatures de fonctions à Implémenter dans la suite du fichier :

```
99
100 void ERROR(CODE_ERR);
101 void sym_suiv();
102 void PROGRAM();
103 void BLOCK();
104 void CONSTS();
105 void VARS();
106 void INSTS();
107 void INST();
108 void AFFEC();
109 void SI();
110 void TANTQUE();
111 void ECRIRE();
112 void LIRE();
113 void COND();
114 void EXPR();
115 void TERM();
116 void FACT();
117 void REPETER();
118 void POUR();
119 void CAS();
120
```

Fonctions pour l'Analyseur Syntaxique :

Cette partie est consacré à l'implémentation des fonctions utiles dans la phase de l'analyse syntaxique.

La fonction **lire_nombre()** permettre la lecture des nombres à l'aide des fonctions **isdigit()** et **isalpha()**.

La fonction **test_symbole()** qui vérifier que le symbole courant est un code lexical, sinon une **erreur** va être détecter.

```

120
121 FILE *file;
122
123 void lire_nombre(){
124     while(isdigit(Car_Cour)){
125         *pchar++ = Car_Cour;
126         Car_Cour = fgetc(file);
127     }
128     if(!isalpha(Car_Cour)){
129         Sym_Cour=NUM_TOKEN;
130     }
131 }
132
133
134 void test_symbole(CODE_LEX cl, CODE_ERR erreur){
135
136     if(Sym_Cour==cl){
137         memset (chainePR, 0, sizeof (chaine));
138         strcpy(chainePR,chaine);
139         // printf("***/////");
140         sym_suiv();
141     }
142     else ERROR(erreur);
143 }
144
145

```

La grammaire du langage pascal :

Dans cette partie on doit inscrire les différentes règles de production de la grammaire.

La table des règles syntaxique :

```

PROGRAM ::=      program ID ; BLOCK .
BLOCK    ::=      CONSTS VARS INSTS
CONSTS   ::=      const ID = NUM ; { ID = NUM ; } | ε
VARS     ::=      var ID { , ID } ; | ε
INSTS    ::=      begin INST { ; INST } end

INST     ::=      INSTS | AFFEC | SI | TANTQUE | ECRIRE | LIRE | ε

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 )

```

Les règles syntaxique En C :

```

146 void PROGRAM(){
147     test_symbole(PROGRAM_TOKEN,ERR_PROGRAM);
148     test_symbole(ID_TOKEN,ERR_ID);
149     test_symbole(PV_TOKEN,ERR_PV);
150     BLOCK();
151     test_symbole(PT_TOKEN,ERR_PT);
152 }
153
154 void BLOCK(){
155     CONSTS();
156     VARS();
157     INSTS();
158 }
159
160 void CONSTS(){
161     switch(Sym_Cour){
162     case CONST_TOKEN: sym_suiv();
163                     test_symbole(ID_TOKEN,ERR_ID);
164                     test_symbole(EG_TOKEN,ERR_EGAL);
165                     test_symbole(NUM_TOKEN,ERR_NUM);
166                     test_symbole(PV_TOKEN,ERR_PV);
167                     while(Sym_Cour==ID_TOKEN){
168                         sym_suiv();
169                         test_symbole(EG_TOKEN,ERR_EGAL);
170                         test_symbole(NUM_TOKEN,ERR_NUM);
171                         test_symbole(PV_TOKEN,ERR_PV);
172                     }break;
173     case VAR_TOKEN:break;
174     case BEGIN_TOKEN: break;
175     default :ERROR(ERR_CONST);break;
176 }
177
178 }
179

```

```

181
182 void VARS(){
183
184     switch(Sym_Cour){
185
186         case VAR_TOKEN: sym_suiv();
187                         test_symbole(ID_TOKEN, ERR_ID);
188                         while(VIR_TOKEN==Sym_Cour){
189                             sym_suiv();
190                             test_symbole(ID_TOKEN, ERR_ID);
191                             }test_symbole(PV_TOKEN, ERR_PV);break;
192
193         case BEGIN_TOKEN: break;
194         default : ERROR(ERR_VAR_BEGIN);break;
195     }
196
197 }
198
199 void INSTS(){
200     test_symbole(BEGIN_TOKEN, ERR_BEGIN);
201     INST();
202     while(Sym_Cour==PV_TOKEN){
203         sym_suiv();
204         INST();
205     }
206
207 }

```

```

208
209 void INST(){
210
211     switch(Sym_Cour){
212
213         case BEGIN_TOKEN: INSTS();break;
214         case ID_TOKEN:  AFFEC();break;
215
216         case IF_TOKEN: SI();break;
217         case WHILE_TOKEN: TANTQUE();break;
218         case WRITE_TOKEN: ECRIRE();break;
219         case READ_TOKEN: LIRE();break;
220         case REPEAT_TOKEN : REPETER();break;
221         case FOR_TOKEN : POUR();break;
222         case CASE_TOKEN : CAS();break;
223     }
224
225
226 void AFFEC(){
227     sym_suiv();
228     switch(Sym_Cour){
229         case AFF_TOKEN: test_symbole(AFF_TOKEN, ERR_AFF);break;
230         default: ERROR(ERR_AFF);break;
231     }
232     EXPR();
233
234
235 void TANTQUE(){
236     test_symbole(WHILE_TOKEN, ERR_WHILE);
237     COND();
238     test_symbole(DO_TOKEN, ERR_DO);
239     INST();
240
241 }

```


Les Erreurs Syntaxique :

Pour chaque symbole on lui associé un code d'erreur et un message d'erreur.

Au début on a déclaré une structure de type **Erreur** qu'on a utilisé dans l'analyse syntaxique :

```

74
43 Erreur Erreurs[]={ERR_CAR_INC, "caractère inconnu." },
44 {ERR_FIC_VID, "fichier est vide"},
45 {ERR_PROGRAM, "Erreur dans la syntaxe PROGRAM"},
46 {ERR_ID, "Erreur dans la syntaxe ID"},
47 {ERR_PV, " ';' manquee "},
48 {ERR_PT, " '.' manquee"},
49 {ERR_EGAL, "Erreur dans la syntaxe '=' "},
50 {ERR_NUM, "Erreur dans la syntaxe NUM"},
51 {ERR_CONST, "Erreur dans la syntaxe CONST"},
52 {ERR_VAR_BEGIN, "Erreur dans la syntaxe VAR BEGIN "},
53 {ERR_CONST_VAR_BEGIN, "Erreur dans la syntaxe CONST VAR BEGIN"},
54 {ERR_BEGIN, "Erreur dans la syntaxe BEGIN"},
55 {ERR_END, "Erreur dans la syntaxe END"},
56 {ERR_AFF, "Erreur dans la syntaxe AFFECTATION"},
57 {ERR_IF, "Erreur dans la syntaxe IF"},
58 {ERR_WHILE, "Erreur dans la syntaxe WHILE"},
59 {ERR_DO, "Erreur dans la syntaxe DO"},
60 {ERR_WRITE, "Erreur dans la syntaxe WRITE"},
61 {ERR_READ, "Erreur dans la syntaxe READ"},
62 {ERR_THEN, "Erreur dans la syntaxe THEN"},
63 {ERR_PO, "Erreur dans la syntaxe '('"},
64 {ERR_PF, "Erreur dans la syntaxe ')'"},
65 {ERR_MOINS, "Erreur dans la syntaxe '-'"},
66 {ERR_PLUS, "Erreur dans la syntaxe '+'"},
67 {ERR_MULT, "Erreur dans la syntaxe '*'"},
68 {ERR_DIV, "Erreur dans la syntaxe '/'"},
69 {ERR_INF, "Erreur dans la syntaxe '<'"},
70 {ERR_SUP, "Erreur dans la syntaxe '>'"},
71 {ERR_INFEG, "Erreur dans la syntaxe '<='"},

72 {ERR_SUPEG, "Erreur dans la syntaxe '>='"},
73 {ERR_COND, "Erreur dans la syntaxe COND"},
74 {ERR_TERM, "Erreur dans la syntaxe TERM"},
75 {ERR_FACT, "Erreur dans la syntaxe FACT"},
76 {ERR_DIFF, "Erreur dans la syntaxe '<>'"},
77 {ERR_ACC, "Erreur dans la syntaxe '{'"},
78 {ERR_EXPR, "Erreur dans la syntaxe EXPR"},
79 {ERR_REPEAT, "Erreur dans la syntaxe REPETER"},
80 {ERR_UNTIL, "Erreur dans la syntaxe UNTIL"},
81 {ERR_FOR, "Erreur dans la syntaxe FOR"},
82 {ERR_CASE, "Erreur dans la syntaxe CASE"},
83 {ERR_INT0, "Erreur dans la syntaxe INTO "},
84 {ERR_DOWNTO, "Erreur dans la syntaxe DOWNTO "},
85 {ERR_DP, "Erreur dans la syntaxe ':'"},
86 {ERR_OF, "Erreur dans la syntaxe OF "},
87 {ERR_AFF, "Erreur dans la syntaxe ':=' "},
88 {ERR_ELSE, "Erreur dans la syntaxe ELSE "},
89 {ERR_INST, "Erreur double ';' "},
90 {ERR_NOTEXISTID, "Erreur ID inconnu"},
91 {ERR_EXISTID, "Erreur ID Deja declarer "},
92 {ERR_CONSTCH, "impossible de modifier un CONST type "},
93 {ERR_PROGD, "impossible d'utiliser PROGRAM "},
94 {ERR_PLEIN, "Erreur de PCODE "},
95 {ERR_AFFECT_ABS, "manque ':='"},
96
97 };


```

Tester l'Analyseur syntaxique :

Pour essayer notre analyseur syntaxique on va lui fournir le code suivant :

```
program test11;
const toto=21; titi=13;
var x,y;
Begin.
{* initialisation de x *}
x:=toto;
read(y);
while x<y do begin read(y); x:=x+y+titi end;
{* affichage des resultas
de x et y *}
write(x);
write(y);
end.
```

Le resultat d'exécution donne : " le programme est correct !! "



```
BRAVO: le programme est correct!!
-----
Process exited after 0.04907 seconds with return value 0
Press any key to continue . . .
```