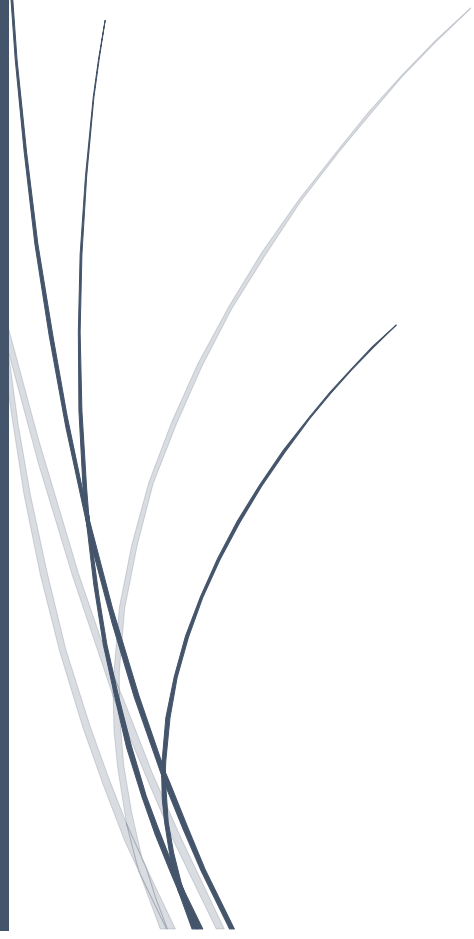
FAIT PAR : MOUAD RIALI — KAMAL ADDI

RAPPORT >

Analyseur Lexical

Langage pascal



ANALYSEUR LEXICAL :

Le processus de compilation d'un programme consiste en un certain nombre d'étapes ; en pratique les compilateurs sont ´écrits pour être capables de les réaliser ensemble, en faisant une seule passe dans sa donnée. Cependant, pour présenter ces étapes séparément permet de mieux comprendre le rôle de chacune de celles-ci.

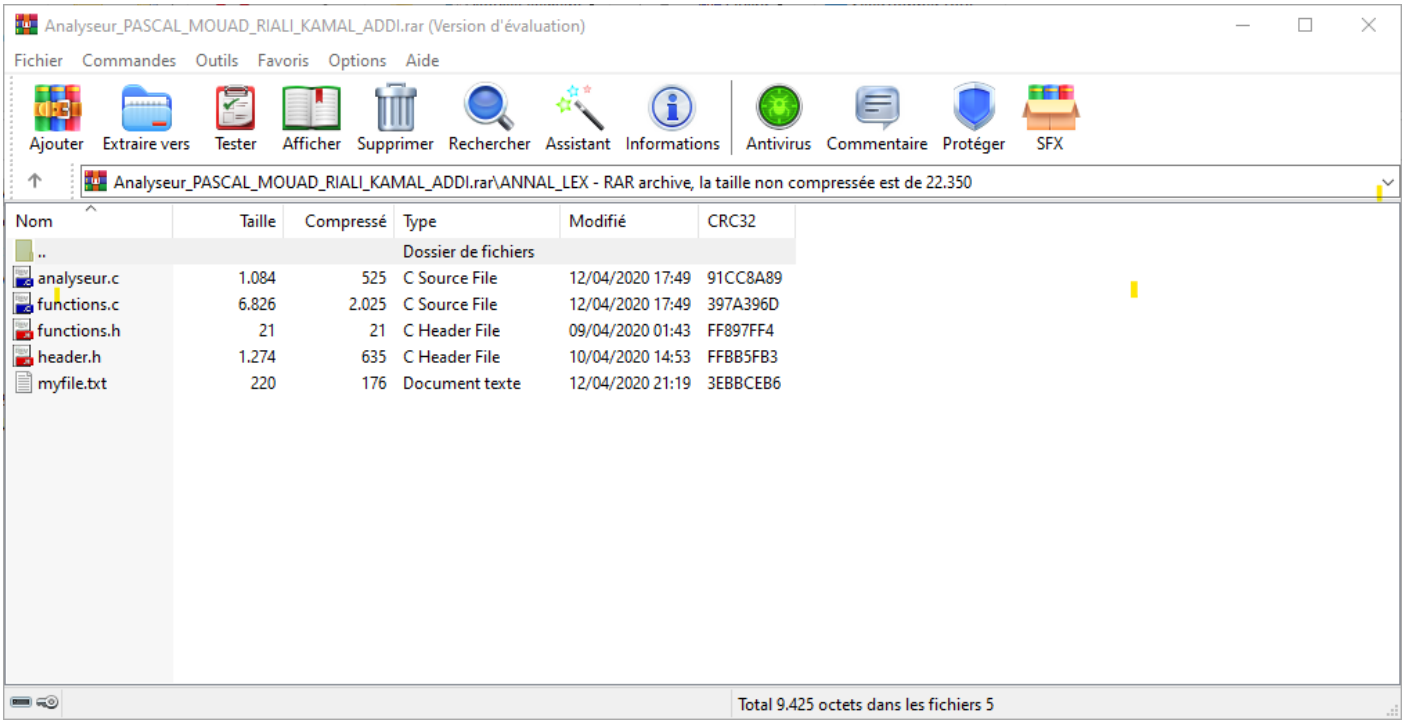
Les trois grandes étapes.
1. L'analyse lexicale ;
2. L'analyse syntaxique ;
3. La production du code objet ;
-> ici on va présenter un analyseur lexical de langage pascal en langage C

. La structure de l'analyseur :

Dans cette étape, on va présenter la forme générale de notre analyseur, c'est-à-dire, on va mentionner et expliquer le rôle de chacune des fichiers existants dans le dossier de l'analyseur.

En effet : notre répertoire se compose de 5 fichiers, 2 parmi eux ont l'extension « .c » : **analyseur.c**, **functions.c** ,2 autres avec l'extension « .h » : **functions.h** et **header.h** et une dernière fichier texte :

Myfile.txt .



« header.h » !!!!

Ce fichier et comme son nom signifie, est le fichier ou il existe l'ensemble des énumérations, structures, des listes et des variables qu'on va surement utiliser dans notre programme main.

Effectivement : on peut décomposer ce fichier en plusieurs parties.

- Les premiers 7 lignes sont réservées à l'appel des bibliothèques de langage C ou même des constantes qui vont être utiles par la suite comme **IDFMAX,** les bibliothèques appelées sont :

  ->**stdbool.h** /**string.h** : il nous permet de manipuler les variables booléennes et chaines de caractères. (strcpy /strcmp/….)

  ->**stdio.h** / **stdlib.h** : le minimum des bibliothèques essentielles pour un code C

  ->**ctype.h** : un header file qui contient des fonctions pour déterminer le type d'un caractère – isalpha( ), isdigit( ) …

```c
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <ctype.h>
#define IDFMAX 70
```

- La deuxième partie est l'ensemble des listes et enum qui vont définir des variables et des nouveaux serviable types par la suite dans la construction des fonctions :

  -> **mot_cles []** : contient une liste des chaines de caractères des mot clés définis par le support de projet.

  -> **symbol_speci []** : de même pour « mot_cles » cette liste contient les symboles spéciaux déclarés dans le support de projet.

  -> **CODE_LEX** : un nouveau type de variable crée à partir de l'énumération au-dessous, il contient les toKens des symboles et des mot clés de ce langage, en respectant l'ordre de positionnement de chaque symbole ou mot clés par rapport les autres symboles et mot clés.

```c
//MOTS CLES :

const char* mot_cles[]={"program","const","var","begin","end","if","then","while","do","read","write"};
//LES SYMBOLES SPECIAUX :

char* symbol_speci[]={";",".","+","-","*","/",",",":=","<","<=",">",">=","<>","(",")","EOF","="};

//L'univers de TOKEN :
typedef enum{
    //Mots cles :
    PROGRAM_TOKEN,CONST_TOKEN,VAR_TOKEN,BEGIN_TOKEN,END_TOKEN,IF_TOKEN,THEN_TOKEN,
    WHILE_TOKEN,DO_TOKEN,READ_TOKEN,WRITE_TOKEN,
     //LES SYMBOLES SPECIAUX :
     PV_TOKEN,PT_TOKEN,PLUS_TOKEN,MOINS_TOKEN,MULT_TOKEN,DIV_TOKEN,VIR_TOKEN,AFF_TOKEN,
     INF_TOKEN,INFEG_TOKEN,SUP_TOKEN,SUPEG_TOKEN,DIFF_TOKEN,PO_TOKEN,PF_TOKEN,FIN_TOKEN,EG_TOKEN,
     //Autres :
     ID_TOKEN,
     NUM_TOKEN,
     ERREUR_TOKEN
}CODE_LEX;
```

- à l'aide des structures ,on déclare ici les majeurs types qu'on va les utiliser pour stocker soit les noms de tokens ou bien la valeur de token -si on a besoin d'elle- comme il est clair au-dessous :

```
31  //structure d'un identificateur :
32
33  typedef struct{
34      char *name;
35  }id_token;
36
37  //structure d'un nombre :
38
39  typedef struct{
40      bool isInt; // entier ou flottant
41      union{
42          int n;
43          float x;
44      }value;
45  }est_num;
46
47
48  // La structure d'un token
49  typedef struct{
50      CODE_LEX name; // Le nom du token
51      union{
52          id_token idf; // les informations de l'id (si le token est ID_TOKEN)
53          est_num number; // les informations du nombre (si le token est NUMBER_TOKEN)
54      }properties;
55  } token;
56
```

functions.h : une seule ligne !!

Pour être clair, cet header file n'est pas assez nécessaire pour que notre programme fonctionne, mais pour des raisons d'organisation et d'esthétiques, on a décidé de faire appel a **functions.h** comme header de **functions.c** au lieu de **header.h**

En effet :

functions.h

```
1   #include "header.h"
```

.

functions.c : un peu de sérieux

Dans ce fichier on va construire les fonctions nécessaires pour l'extraction des Token et la détermination de leurs types selon l'enum **CODE_LEX** défini dans **header.h**

En Pratique :

On va appeler tout d'abords les variables, les bibliothèques, les énumérations, les listes nécessaires pour achever notre objectif :

```c
#include "functions.h"

bool isNumber=true;
int mots_cles_size=sizeof(mot_cles)/sizeof(mot_cles[0]);
int sym_speci_size=sizeof(symbol_speci)/sizeof(symbol_speci[0]);


bool estmotcle(char *c);

int symbole=0;

```

- Cette fonction permet de lire un caractère nombre a l'aide de boucle **do---while** et la fonction **isdigit()** pour nous donner en fin un output de type **est _num** (voire header.h ->3)

```c
est_num getNumber(char digit, bool isNegative){
    int i=0;
    // variable qui indique s'il s'agit d'un entier ou un flottant
    bool isInt = true;
    // Allocation de la mémoire
    char* memory = (char *)malloc(NUMBERMAX*sizeof(char));
    est_num A;
    // Lire tout le chiffre
    do{
        memory[i] = digit;
        digit = fgetc(fichSrc); // caractère suivant
        i++;
        if(digit == '.'){ // si on arrive à une virgule (un nombre flottant)
            isInt = false;
            memory[i] = '.';
            digit = fgetc(fichSrc);
            i++;
        }
    }while(isdigit(digit)!= 0);
    memory[i] = '\0';
    /*
        on sort de do..while Lorsqu'on lit un caractère non numérique
        il faut retourner le curseur pour que cette caractère ne soit perdu
    */
    ungetc(digit, fichSrc);
    A.isInt = isInt;
    if(isInt == true){ // un nombre entier
        // transformer le nombre dans la chaine de caractère memory en int et le stoker dans A.value.n
        A.value.n = atoi(memory);
        if(isNegative == true){
            A.value.n = - A.value.n;
        }
    }else{ // un nombre flottant
        // transformer le nombre dans la chaine de caractère memory en float et le stoker dans A.value.x
        A.value.x = atof(memory);
        if(isNegative == true){
            A.value.x = - A.value.x;
        }
    }
    return A;
}
```

Détails pratiques : l'utilité de cette fonction c'est de faciliter la tâche quand on a le cas des **NUM_TOKEN** dans la fonction suivante **get_token()** qu'on va la deviser en plusieurs parties due a sa grande taille et a la diversité de ses objectifs.

- La première partie c'est la déclaration des variables locales, l'élimination des blancs et la lecture du caractère suivant :

```
token getToken(){
    bool previousIsNumber = isNumber;
    isNumber = false;
    // Lire le caractère suivant
    char character = fgetc(fichSrc);
    token A;
    int i = 0;
    // Eliminer les blancs
    if(character == ' ' || character == '\t' || character == '\n'){
        return getToken();
    }
```

- La deuxième c'est la Reconnaissance des mots-clés et des identificateurs :

```
    // Reconnaissance des mots-clé et des identificateurs
    else if(isalpha(character) != 0){
        // Allocation de la mémoire pour sauvegarder le mot
        A.properties.idf.name = (char *)malloc(IDFMAX*sizeof(char));
        char *mot=(char *)malloc(IDFMAX*sizeof(char));
        // Lire tout le mot
        do{
            mot[i]=character;
            character=fgetc(fichSrc);
            i++;
        }while((isalpha(character)!=0) || (isdigit(character)!=0));
        strcpy(A.properties.idf.name,mot); // il faut poser \0 à la fin du mot
        ungetc(character, fichSrc);
        i=0;
        // vérifier si lo mot obtenue est un mot-clé :
        while(i < mots_cles_size){
            if(stricmp(A.properties.idf.name, mot_cles[i]) == 0){ // Le mot est un mot-clé
                // Le nom token c'est (nameToken) i
                A.name = (CODE_LEX) i;
                // Libérer la case mémoire A.properties.idf.name
                // On a besoin du nom de token seulement dans ce cas (et pas sa valeur)
                free(A.properties.idf.name);
                return A; // retourner le token
            }
            i++;
        }
        // Si on arrive à cette étape, le mot n'est pas mot-clé
        // Donc c'est un identificateur, et son nom est stocké dans A.properties.idf.name

        A.name = ID_TOKEN;
        return A;
    }
```

- La reconnaissance de Nombres :

```
    // Reconnaissance des nombres
    else if(isdigit(character) != 0){
        isNumber = true;
        A.name = NUM_TOKEN;
        // récupérer le nombre et le stocker dans A.properties.number
        A.properties.number = getNumber(character, false);
        return A;
    }
    else if(character == '+' || character == '-'){
        CODE_LEX symboleToken = (character == '+' ? PLUS_TOKEN : MOINS_TOKEN);
        bool isNegative = (character == '-' ? true : false);
        // Eliminer les espaces
        do{
            character = fgetc(fichSrc);
        }while(character == ' ');
        // Si le caractère suivant de +/- est un chiffre
        if(isdigit(character) != 0){
            if(previousIsNumber == true){
                // le token précédant est un nombre, exemple de cette situation : "5 - 3"
                ungetc(character, fichSrc); // retourner le curseur en arrière
                A.name = symboleToken; // token : PLUS_TOKEN ou MOINS_TOKEN
                return A;
            }else{
                // le token précédant n'est pas un nombre, exemple cette situation : "= - 3"
                isNumber = true; // token actuel est un nombre
                A.name = NUM_TOKEN;
                A.properties.number = getNumber(character, isNegative);
                return A;
            }
        }else{ // Le caractère suivant de +/- n'est pas un chiffre
            ungetc(character, fichSrc); // retourner le curseur en arrière
            A.name = symboleToken;
            return A;
        }
    }
```

- Et pour ne pas être dérangés par les commentaires surtout qu'ils n'influencent pas sur le code, on peut les éliminer définitivement par ce bout de code :

```
    // Elimination des commentaires
    else if(character == '{'){
// Lire le caractère suivant
        character = fgetc(fichSrc);
        if(character == '{'){ // un commentaire ligne
            // dépasser tous les caractères jusqu'on arrive à \n
            do{
                character = fgetc(fichSrc);
            }while(character != '\n' && character != EOF);
            return getToken(); // rappeler la fonction
        }
        else if(character == '*'){ // un commentaire bloque
            // dépasser tous les caractères jusqu'on arrive à */
            do{
                character = fgetc(fichSrc);
            }while(character != '*');
            // Lire le caractère suivant
            character = fgetc(fichSrc);
            if(character == '}'){ // Fin du commentaire bloque
                return getToken(); // rappeler la fonction
            }
        }

    }
```

- Passons ensuite a la reconnaissance des symboles simples :

```
163     // Reconaissance des symboles simples
164     else if(character==';'){
165         A.name = PV_TOKEN;
166         return A;   }
167     else if(character=='.'){
168         A.name = PT_TOKEN;
169         return A;   }
170     else if(character=='*'){
171         A.name = MULT_TOKEN;
172         return A;
173     }
174     else if(character=='/'){
175         A.name = DIV_TOKEN;
176         return A;
177     }
178     else if(character==','){
179         A.name = VIR_TOKEN;
180         return A;
181     }
182     else if(character=='('){
183         A.name = PO_TOKEN;
184         return A;
185     }
186     else if(character==')'){
187         A.name = PF_TOKEN;
188         return A;
189     }
190     else if(character=='='){
191         A.name = EG_TOKEN;
192         return A;
193     }
```

- La Reconnaissance des symboles complexes : <=, :=,>=

```
195     Reconnaissance des symboles complexes
196     */
197     // Reconnaissance de <, <>,<=
198     else if(character == '<'){
199         // Lire le caractère suivant
200         character = fgetc(fichSrc);
201         if(character == '>'){
202             A.name = DIFF_TOKEN;
203             return A;
204         }else if(character=='='){
205             A.name = INFEG_TOKEN;
206             return A;
207         }else{
208             A.name = INF_TOKEN;
209             ungetc(character, fichSrc);
210             return A;
211         }
212     }
213     else if(character=='>'){
214         character = fgetc(fichSrc);
215         if(character=='='){ // On a le symbole >=
216         A.name = SUPEG_TOKEN;
217         return A;}
218         else{
219         A.name = SUP_TOKEN;
220         ungetc(character, fichSrc);
221         return A;
222         }
223     }
224     // Reconnaissance de >, >=
225     else if(character == ':'){
226         // Lire le caractère suivant
227         character = fgetc(fichSrc);
228         if(character == '='){ // On a le symbole :=
229             A.name = AFF_TOKEN;
230             return A;
231         }else{
232             A.name = ERREUR_TOKEN;
233             ungetc(character, fichSrc);
234             return A;
235         }
```

- Enfin, il nous reste d'étudier le cas ou on a soit la fin de fichier (EOF) ou un caractère qui n'est pas reconnu dans le langage :

```
237     // La fin de la lecture
238     else if(character == EOF){
239         A.name = FIN_TOKEN;
240         return A;
241     }
242     // Si on rencontre d'autre caractères non analysés
243     else{
244         A.name = ERREUR_TOKEN;
245         return A;
246     }
247 }
```

analyseur.c : main code

Ce fichier est une synthèse de tout le travail évoqué avant, car c'est ici ou on va utiliser et mettre en ordre tout nos variables et fonctions pour qu'on recoit le résultat souhaité, d'abords pour l'input, on a posé un chemin directif vers le fichier qui contient le code, puis et après le traitement des données on va faire apparaitre le résultat dans le terminal, ainsi on va l'enregistrer dans un autre fichier texte pour la conserver et l'enregistrer.

Les parties de code : on a 2 parties majeures :

- La partie de déclaration des variables, des biblio et des header files :

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4
5   FILE *fichSrc;
6   FILE *f;
7
8   #include "functions.c"
9   const char* tokens[] = {    //Mots cles :
10      "PROGRAM_TOKEN","CONST_TOKEN","VAR_TOKEN","BEGIN_TOKEN","END_TOKEN","IF_TOKEN","THEN_TOKEN",
11      "WHILE_TOKEN","DO_TOKEN","READ_TOKEN","WRITE_TOKEN",
12       //LES SYMBOLES SPECIAUX :
13       "PV_TOKEN","PT_TOKEN","PLUS_TOKEN","MOINS_TOKEN","MULT_TOKEN","DIV_TOKEN","VIR_TOKEN","AFF_TOKEN",
14       "INF_TOKEN","INFEG_TOKEN","SUP_TOKEN","SUPEG_TOKEN","DIFF_TOKEN","PO_TOKEN","PF_TOKEN","FIN_TOKEN","EG_TOKEN",
15       //Autres :
16       "ID_TOKEN",
17       "NUM_TOKEN",
18       "ERREUR_TOKEN"};
```

Les deux premières lignes sont écrites pour pouvoir déclarer les deux pointeurs de type FILE

« **fichSrc** » et « **f** »,

On a appelé le fichier « functions.c » car il contient toutes les fonctions qu'on a besoin ainsi que les bibliothèques nécessaires pour l'assurance du bon déroulement de processus de l'analyseur syntaxique.

La liste « **tokens** » va permettre au programme de faire apparaitre ou enregistrer le résultat de l'analyse soit au niveau de terminal ou au niveau de fichier.

- Le main code :

```
19  int main(){
20      //Fichier source :
21      fichSrc=fopen("myfile.txt", "r");
22      f=fopen("myfile_output1.txt","w");
23      if(fichSrc==NULL){
24          printf("Chemin non valide !!");
25          exit(0);
26      }
27          int k;
28      token currentToken = getToken();
29      do{
30          k = (int) currentToken.name;
31          printf("%s ", tokens[k]);
32          fprintf(f,"%s\n",tokens[k]);
33          currentToken = getToken();
34      }while(currentToken.name != FIN_TOKEN);
35      fprintf(f,"FIN_TOKEN");
36      printf("\nFIN_TOKEN");
37      return 0;
38  }
39
```

On va en premier temps déterminer un chemin pour accéder au fichier qui contenir le code à analyser, après, il faut s'assurer si ce chemin est valide ou non avant commencer pour ne pas avoir des problèmes de compilation (23 <> 26)

On va initialiser le variable currentToken de type token par la première valeur retournée par la fonction **getToken()** .

Faisons une boucle **do—while** et écrire le résultat soit dans le terminal **print** ou bien dans le fichier **fprintf()** en liant entre le rang de token retourné par getToken() et le rang de chaine de caractères dans la liste tokens ,

Et voilà : lorsqu'on arrive à **FIN_TOKEN** le programme va sortir de la boucle et enregistrer l'analyse lexicale dans un autre fichier totalement séparé du premier fichier qui contient le code à analyser

PS : L'ordre ici est axiale, alors il faut faire attention de suivre le même ordre soit dans l'enum, ou la liste des symboles, ou la dernière liste tokens.