Créer un compilateur pour votre propre langage

19 Février 2022

Par:
NAIM Kawtar

https://sdadclub.tech/

Ce cours est assuré par :

NAIM Kawtar

étudiante en master Sciences des données et aide à la décision à l'université Cadi Ayyad et une membre du club SDAD.

kawtarnaim1999@gmail.com

https://www.linkedin.com/in/kawtar-naim-16bbaa151/

Plan :

Ch. 1 : Introduction Générale

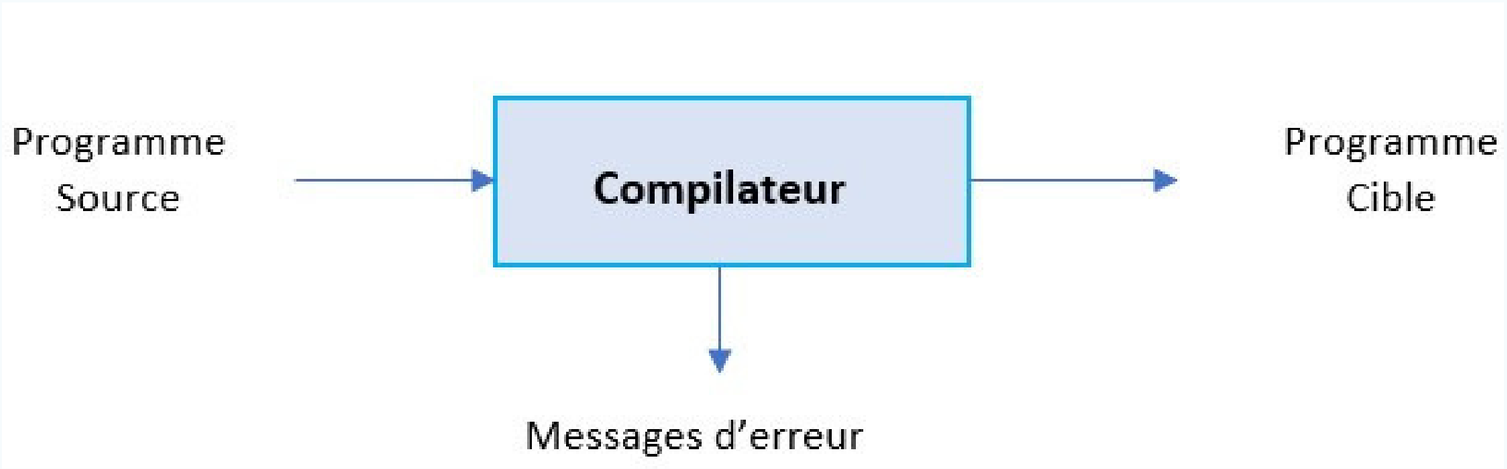
Pourquoi étudier la compilation ?

- C'est une des branches les plus dynamiques de l'informatique, et une des plus anciennes à mériter ce qualificatif.

- Son domaine d'application est très large : conversion de formats ; traduction de langages ; optimisation de programmes.

Qu’est-ce qu’la compilation ?

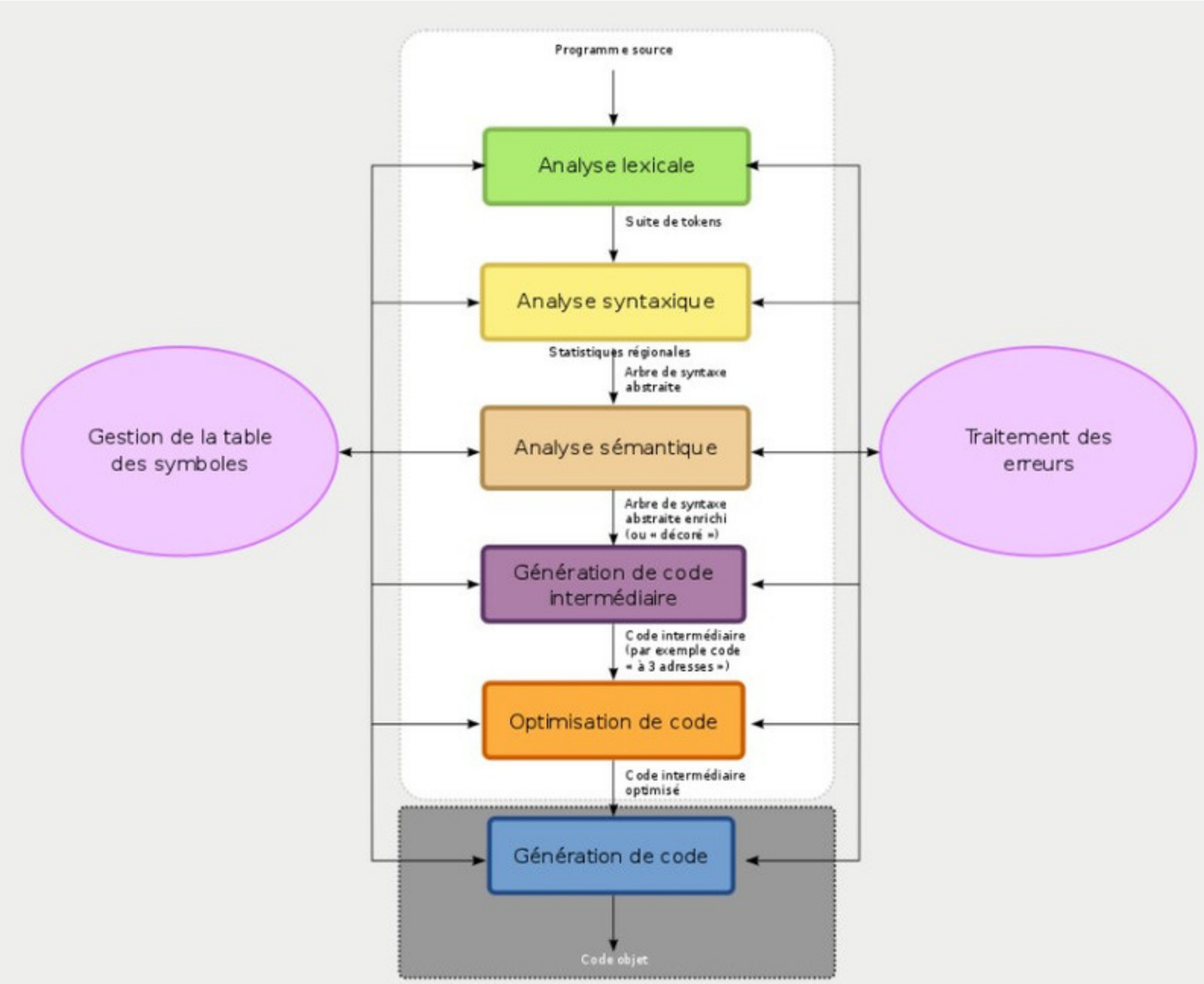
C'est une traduction : un texte écrit en C, C++, Java, etc., exprime un algorithme et il s'agit de produire un autre texte, spécifiant le même algorithme dans le langage d'une machine que nous cherchons à programmer



Quelques qualités d'un compilateur

- Confiance – S'approcher du « zéro bug »
- Optimisation – Code généré rapidement
- Efficacité – Compilateur rapide
- Economie – Compilateur portable
- Productivité – Erreurs signalées utiles

Phase d’un compilateur



Ch. 2 : Analyse Lexicale

L'analyse lexicale représente la première phase de la compilation.

Rôle d'un analyseur lexicale (scanner) : lire les caractères d' entrée (programme source) et produit comme résultat une suite d'unités lexicales appellées **tokens** que l' analyseur syntaxique va utiliser.

Les analyseurs lexicaux assurent aussi certaines fonctionnes telles que :

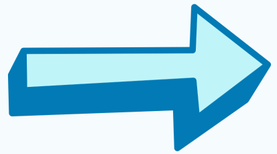
- La suppression des caractères de décoration (blancs, indentations, fins de ligne, etc.) et celle des commentaires (généralement considères comme ayant la même valeur qu'un blanc),

- La gestion des fichiers et l'affichage des erreurs, etc

TOKENS

Code 1

```
Début {
      ma_Varaibale=12;
      b=2;
      si(var1!=b) alors{
            var1 = b;
      }
fin
```



Début	**DEBUT**
(	**PO**
)	**PF**
ma_variable b var1	**IDENTIF**
=	**EGAL**
12 2	**ENTIER**
:	**PV**
si	**SI**
!=	**DIFF**
alors	**ALORS**
{	**AO**
}	**AF**
fin	**FIN**

une éxpression régulière

une éxpression régulière est une chaîne de caractères qui décrit un ensemble (fini ou infini) de chaînes de caractères.

Nouveaux concepts :

Alphabet

{0,1}, {A, C, G, T}, L’ensemble de toutes les lettres, L'ensemble des chiffres etc.

Mot

chaine des alphabets EX 01010101 SDAD ...

Langage

L 'ensemble des chaines ADN, L'ensemble des mots de la langue français

Opérations sur les mots

Concaténation

La concaténation de deux chaînes x et y donne lieu au mot xy

EX : **Club** concaténé avec **sdad** donne Clubsdad

Puissance

- $x^0 = \varepsilon,$ $x^n = xx^{n-1}$ si $n > 0$

Opérations sur les langages

Union

Dénomination	Notation	Définition
Union de L et M	L **U** M	{x | x de L **ou** x de M}

Exemple: Soient Σ= {a,b,c}, L_1 ={a,b} et L_2 ={bac, b, c} : **L_1 U L_2 = {a, b, bac, c}**

Concaténation

Dénomination	Notation	Définition
Concaténation de L et M	LM	{xy | x de L **et** y de M}

Exemple: Soient Σ= {a,b,c}, L_1 ={a,b} et L_2 ={bac, b, a} : **L_1L_2 = {abac, ab, aa, bbac, bb, ba}**

Fermeture de Kleene

Dénomination	Notation	Définition
Fermeture de **Kleene** de L	L^*	$\{x_1x_2...x_n \mid x_i$ de L, i de N et i >=0$\}$

Fermeture positive

Dénomination	Notation	Définition
Fermeture positive de L	L^+	$\{x_1x_2...x_n \mid x_i$ de L, i de N et i >=1$\}$

- Exemple: Soient Σ= {a}, L = {a}
  - L^0 ={ε}, L^1={a}, L^2 ={aa}, L^3={aaa},…,
  - L^+ = {a,aa,aaa,aaaa,aaaaa,…..},

Exemple de quelques définitions régulières qui définissent les identificateurs et nombres du langage C

Lettre → A | B | … | Z | a | b | … | z devient lettre → [A-Za-z]

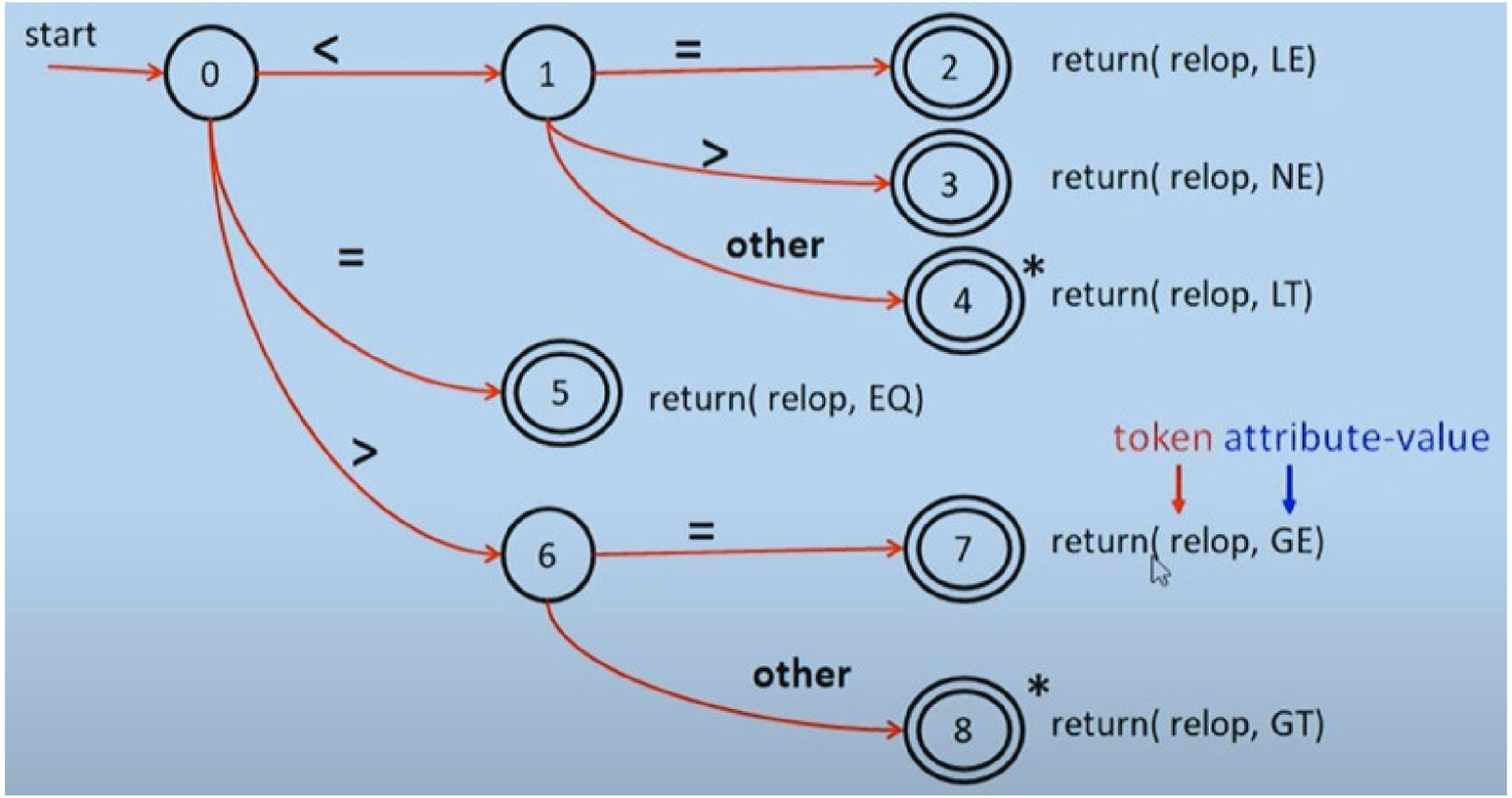
Chiffre → 0 | 1 | … | 9 devient chiffre → [0-9]

Identificateur → lettre (lettre | chiffre | _) devient identificateur → [A-Za-z]([A-Za-z]|[0-9]| _)

Diagrammes de transition

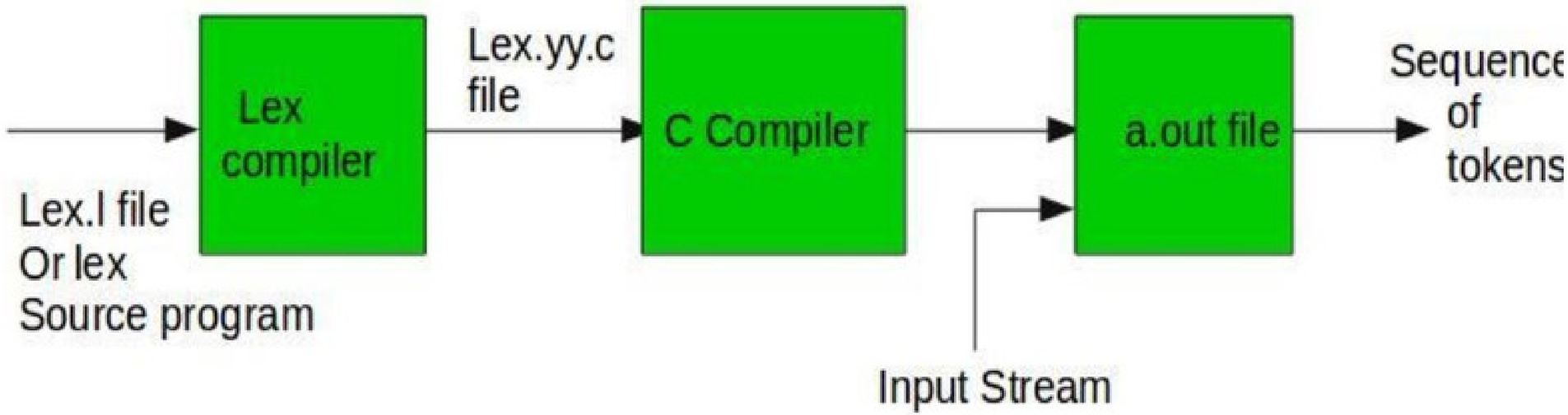
l'**analyse lexicale** utilise un **diagramme de transition** pour garder une trace des informations sur les caractères qui sont vus lorsque le pointeur vers l'avant analyse l'entrée.

EX

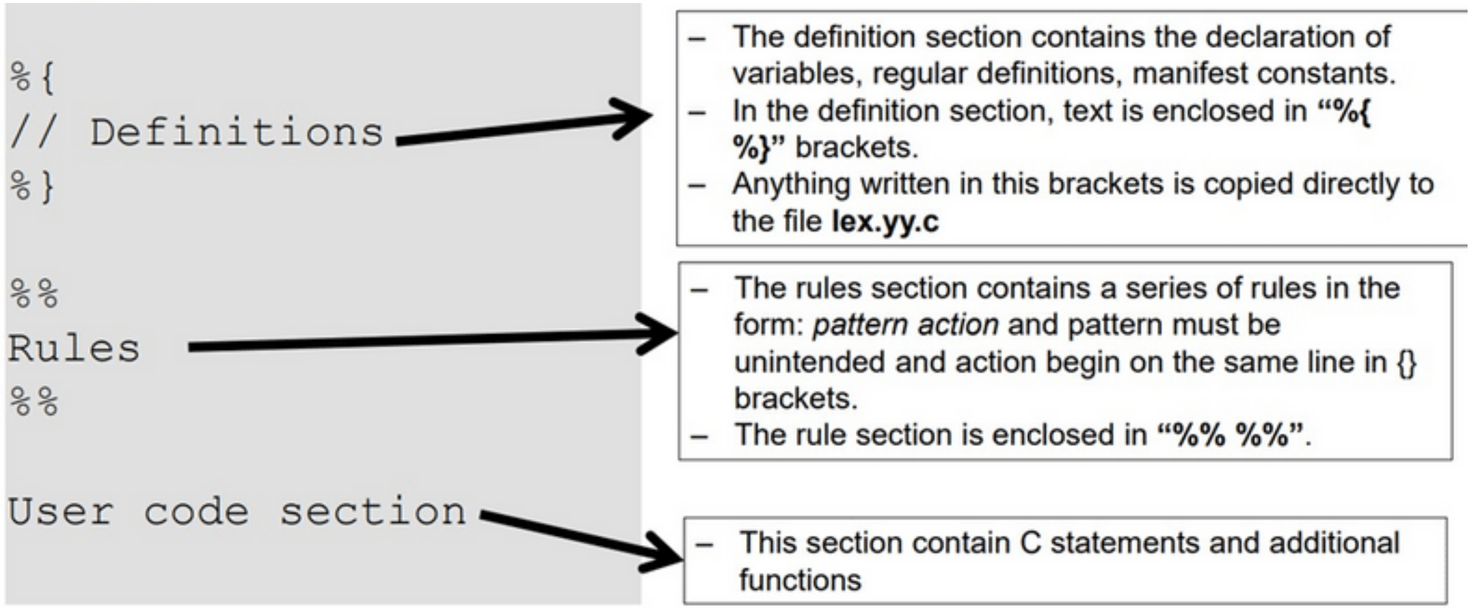


Générateurs d’analyseurs lexicaux

- Les analyseurs lexicaux basés sur des tables de transitions sont les plus efficaces.
- Malgré la construction de cette table est une opération longue et délicate.
- Le programme lex (flex) permet de faire cette construction automatiquement.



Structure de programe flex



- Pour exécuter le programme, il doit d'abord être enregistré avec l'extension .l

Les commandes d ' exécutions :

```
$ flex -omonCompilateur.c monFichier.l
$gcc monCompilateur.c monFichier.Principal.c -o monCompilateur
$ monCompilateur < programme.txt
```

Pratique

C: > Users > kawtar > Desktop > c

```
1   Debut
2   ma_Varaibale=12;
3   b=2;
4   si(var1!=b) alors{
5   var1 = b;
6   }
7   fin
```

Code

unitesLexicales.h × code.txt

C: > Users > kawtar > Desktop > doc > S6

```
1   #define DEBUT 300
2   #define FIN 301
3   #define ENTIER 302
4   #define IDENTIF 303
5   #define SI 270
6   #define ALORS 271
7   #define DIFF 281
8   #define EGAL 282
9   #define PO 283
10  #define PF 284
11  #define AO 285
12  #define AF 286
13  #define PV 287
14  extern int valEntier;
15  extern char valIdentif[];
```

tokens (* .l)

C: > Users > kawtar > Desktop > doc > S6 > Compilation > td1 > code1.l

```
1   %{
2   #include <string.h>
3   #include "unitesLexicales.h"
4   %}
5   nbr [0-9]
6   entier {nbr}+
7   identif [a-zA-Z_][0-9a-zA-Z_]*
8   %%
9   debut { ECHO; return DEBUT; }
10  fin { ECHO; return FIN; }
11  {entier} { ECHO; valEntier = atoi(yytext); return ENTIER; };
12  {identif} {ECHO; strcpy(valIdentif, yytext); return IDENTIF; }
13  [\n] {}
14  [\t] {}
15  [" "] {}
16  "(" {ECHO; return PO;}
17  ")"  {ECHO; return PF;}
18  "{"  {ECHO; return AO;}
19  "}"  {ECHO; return AF;}
20  ";"  {ECHO; return PV;}
21  "!=" {ECHO; return DIFF;}
22  "="  {ECHO; return EGAL;}
23  .    { ECHO; return yytext[0]; }
24  %%
25  int valEntier;
26  char valIdentif[256];
27  int yywrap(void) {
28   return 1 ;
29  }
```

fichier.l

Chp3 L'analyse syntaxique

L'analyse syntaxique est l'une des opérations majeures d'un compilateur qui consiste à indiquer si un texte est grammaticalement correct et à en tirer une représentation interne, que l'on appelle arbre syntaxique.

Grammaire

Tout langage de programmation possède des règles qui indiquent la structure syntaxique d'un programme bien formé.

EX:

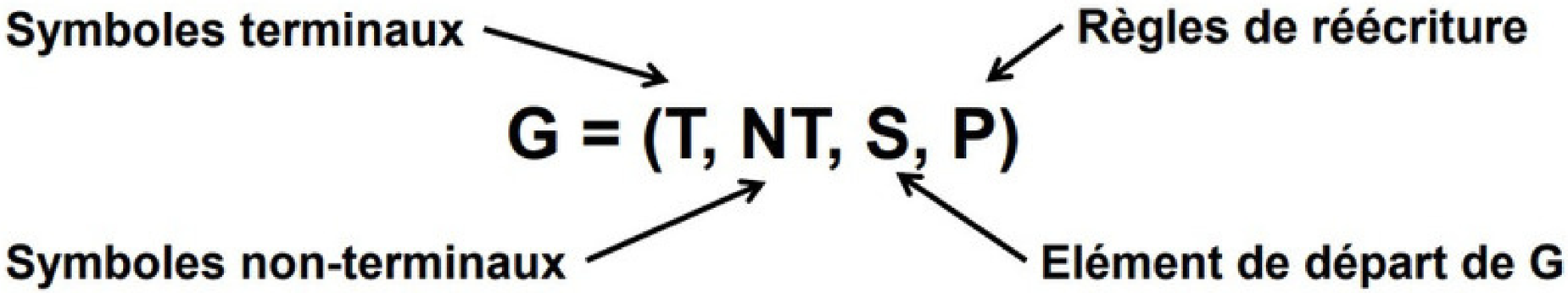
- **If (X > 0) { a = X + 2 ; }**

On distingue :

- Les symboles terminaux : les lettres du langage (if, { , })
- Les symboles non terminaux : les symboles qu'il faut encore définir (ceux en rouge dans l'exemple précèdent)

Grammaire

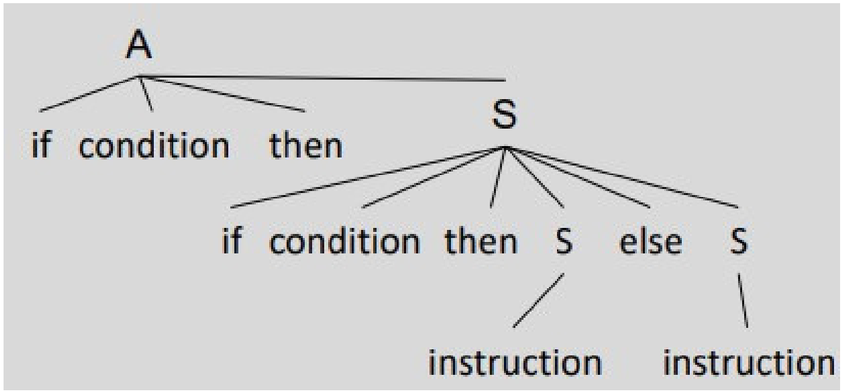
Une grammaire hors contexte (context free grammar, CFG) est un quadruplet



Pratique

Donner la suite de dérivation pour générer le mot **ω** = if a then if b then c else d.

- Générer l' arbre syntaxique



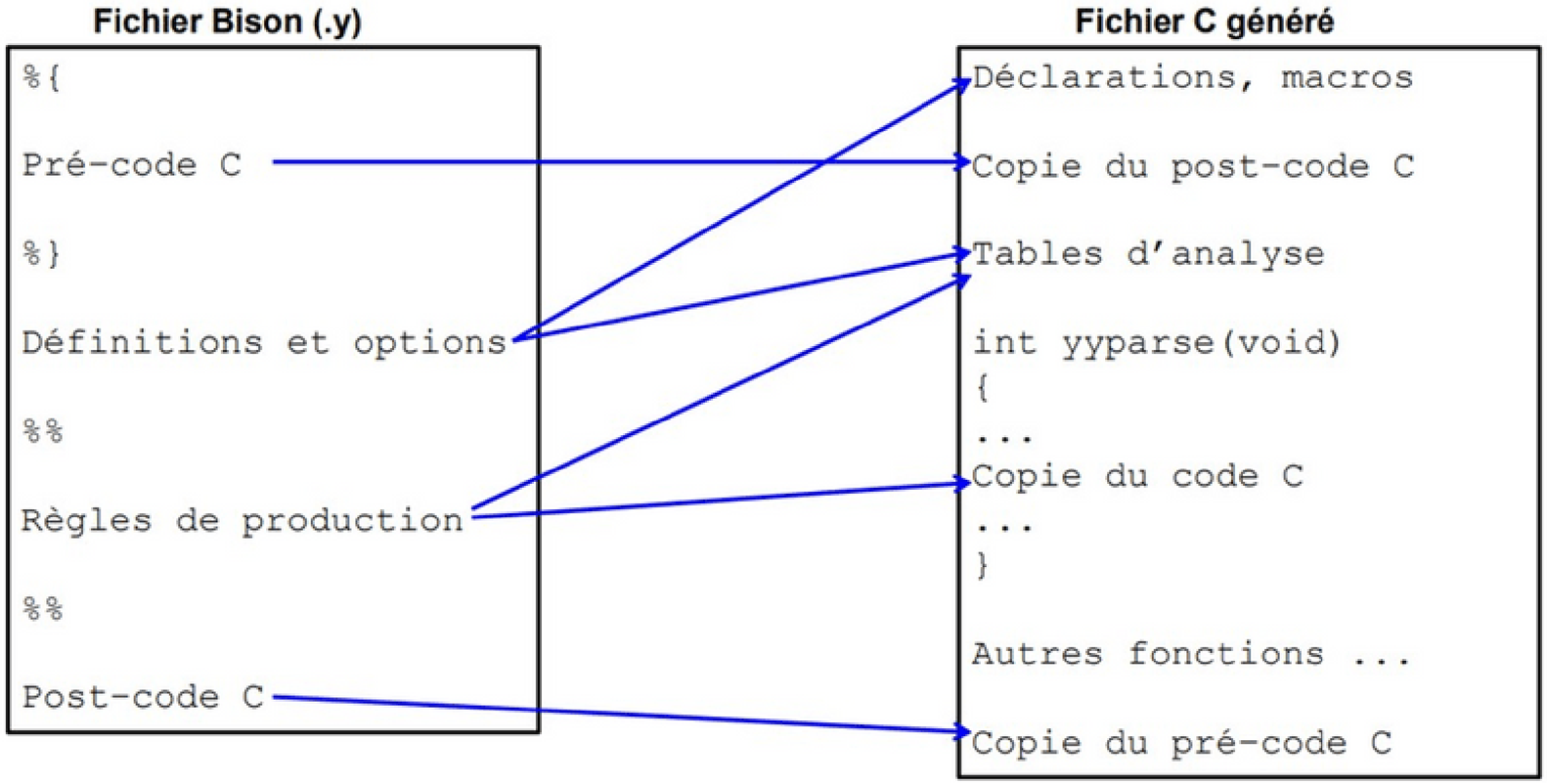
- Alors la grammaire de mot w est :

1) A→ if condition then S
2) S→if condition then S else S
3) S → instruction

Un générateur d'analyseurs syntaxiques

Yacc/Bison : Yet Another Compiler Compiler – Reconnaisseur de langages non contextuels – Grammaire non contextuelle → code C d'un analyseur syntaxique – Permet de reconnaitre les phrases d'un langage.

Structure d'un programme Bison



Pratique

C: > Users > kawtar >

```
1  debut
2  Var = 1;
3  c = Var;
4  a=4 ;
5  fin
```

code

C: > Users > kawtar > Desktop > doc > S6 > Comp

```
1   %{
2   extern int lineNumber;
3   #include "syntaxeY.h"
4
5   %}
6   %option noyywrap
7   nbr [0-9]
8   entier {nbr}+
9   identif [a-zA-Z_][0-9a-zA-Z_]*
10  %%
11  debut { return DEBUT; }
12  fin { return FIN; }
13  [" "\t] { /* rien */ }
14  {identif} { return IDENTIF; }
15  {entier} { return ENTIER; }
16  "=" { return AFFECT; }
17  ";" { return PTVIRG;}
18  "\n" { ++lineNumber; }
19  . { return yytext[0]; }
20  %%
```

fichier . l

C: > Users > kawtar > Desktop > doc > S6 > Compilation > td2 > syntaxe.y

```
1   %{
2   #include <stdio.h>
3
4   extern FILE* yyin; //file pointer by default points to terminal
5   int yylex(void); // defini dans progL.cpp, utilise par yyparse()
6   void yyerror(const char * msg);
7
8   int lineNumber; // notre compteur de lignes
9   %}
10
11  %token DEBUT FIN // les lexemes que doit fournir yylex()
12  %token IDENTIF ENTIER AFFECT PTVIRG
13
14  %start program // l'axiome de notre grammaire
15  %%
16  program : DEBUT listInstr FIN {printf(" sqlt pgme\n");}
17  ;
18
19  listInstr : listInstr inst
20            | inst
21  ;
22  inst : IDENTIF AFFECT expr PTVIRG {printf(" instr affect\n");}
23  ;
24  expr : ENTIER {printf(" expr entier\n");}
25       | IDENTIF {printf(" expr identif n");}
26  ;
27  %%
28  void yyerror( const char * msg){
29      printf("line %d : %s", lineNumber, msg);
30  }
31  int main(int argc,char ** argv){
32      if(argc>1) yyin=fopen(argv[1],"r"); // check result !!!
33      lineNumber=1;
34      if(!yyparse())
35          printf("Expression correct \n");
36          return(0);
37  }
```

fichier . y

Etapes de la construction

1. bison -d –o syntaxeY.c syntaxe.y :

Produit le code C syntaxeY.c depuis le fichier Bison syntaxe.y – Option -d pour générer le .h syntaxeY.h

2. flex –o lexiqueL.c lexique.l :

Produit le code C lexiqueL.c depuis le chier Flex lexique.l – Le pré-code C doit inclure syntaxeY.h

3. gcc –o prog lexiqueL.c syntaxeY.c :

Créer l'exécutable (prog.exe)

4. Prog < code.txt :

Analyser la syntaxe du fichier code.txt

CHP. 4 : Analyse sémantique

- Après l'analyse lexicale et l'analyse syntaxique, l'étape suivante dans la conception d'un compilateur est l'analyse sémantique dont la partie la plus visible est le contrôle de type.

- Quelques tâches liées au contrôle de type sont :

– Construire et mémoriser des représentations des types définis par l'utilisateur, lorsque le langage le permet

– Traiter les déclarations de variables et fonctions et mémoriser les types qui leur sont appliqués

– Vérifier que toute variable référencée et toute fonction appelée ont bien été préalablement déclarées

– Vérifier que les paramètres des fonctions ont les types requis

– Contrôler les types des opérandes des opérations arithmétiques et en déduire le type du résultat

– etc.

L' analyse sémantique

```
1  {
2  var
3  variable;
4  var a,b,c;
5  a=4 ;
6  }
7
```

Code

1. La déclaration des tableaux tokens et vars

```
    char vars[20][20];
    char tokens[8][8]={"{","}","var","=",",",";"};
```

2. La Fonction void createIdentif (char *s,char t[][20]) ;

```
void createIdentif(char *s,char t[][20]){
    strcpy(t[i++],s);
}
```

Permet d'ajouter l'identifiant au dictionnaire à la suite d'une instruction de déclaration.

L' analyse sémantique

3. La Fonction int isToken(char *s,char t[][8]);

```
int isToken(char *s,char t[][8]){
    int j;
    for(j=0;j<8;j++){
        if(strcmp(s,t[j])==0 ){
            return 1;
        }
    }
    return 0;
}
```

vérifier si le programmeur a déclaré un token comme identifiant.

4. La Fonction int isDeclared (char *s,char t[][20]);

```
int isDeclared(char *s,char t[][20]){
    int j;
    for(j=0;j<20;j++){
        if( strcmp(s,t[j]) == 0){
            return 1;
        }
    }
    return 0;
}
```

permet de vérifier si l'identifiant est déjà déclaré

L' analyse sémantique

5. La Fonction void isUsable (char *s,char t[][20]);

```
void isUsable(char *s,char t[][20]){
        if(isDeclared(s,t) == 0){
            yyerror("vous utilisez une variable non declare !!!! ");
            exit(-1);

        }
}
```

permet de vérifier si le programmeur utilise une variable non déclarée.

6. void yyerror(const char * msg);

```
void yyerror( const char * msg){
    printf("line %d : %s", lineNumber, msg);
}
```

pour afficher les messages d'erreur

CHP. 5 génération d’un code

Le but de génération d’un code

Pour écrire un compilateur est mieux de passer par un langage intermédiaire.
Un Langage intermédiaire est plus proche de la machine, et dans mon compilateur je veux utiliser Langage C comme un langage intermédiaire pour générer un code .c et rendre exécutable.

génération d’un code

```
1  Algorithme Somme
2  Var A, B, S: Entier
3  Debut
4  A=2 ;
5  B=3 ;
6  S=A+B ;
7  Ecrire (S) ;
8  Fin
```

code

void open(char *s) : Pour ouvrir le fichier c avec le nom de l 'algorithme

```
void open(char *s){
 char tab[30];strcpy(tab,s);strcat(tab,".c");
 F = fopen(tab,"w");
 fprintf(F,"#include<stdio.h>\n#include<stdlib.h>\n#include<string.h>\n");}
```

génération d’un code

void close() : Pour fermer le fichier c

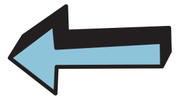
```
void close(){
fprintf(F,"\n/****                    La   FIN                    *** */ \n}");
}
```

void methode_main() : Saisir la fonction main dans le fichier .c

```
void methode_main(){
        fprintf(F," /****   fonction main ********* */\n");
        fprintf(F,"void main(){ \n");}
```

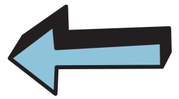
génération d'un code

```
void saisir(char *s,chard[][20]){
strcpy(d[k++],s);strcpy(vars[i++],s);}
```



Copier les variables dans le fichier .c

```
void declare(char *s){fprintf(F,"%s ",s);
for(i=0;i<=20;i++){   fprintf(F,tab[i]);} fprintf(F,";\n");
if(strcmp(s,"int")==0){
 for(i=0;i<20;i++){  saisir(tab[i],inte);}
}
else{   if(strcmp(s,"float")==0){
     for(i=0;i<20;i++){ saisir(tab[i],floate);}
 }
 }
```



Déclarer les variables et presiser leur types

génération d’un code

```
void vide(char tab[][20]){
 for(i=0;i<20;i++){
  strcpy(tab[i],"");
}
```

Permer de librer les variables declarées

```
void cherche(char * s){
 int i;
for(i=0;i<20;i++){  if(strcmp(s,inte[i])==0 ){
        fprintf(F,"%cd",'%');
    }
          if(strcmp(s,floate[i])==0 ){
        fprintf(F,"%cf",'%');
    }}}
```

Permer de lchercher le type des variables declarées

génération d’un code

lexique.l | syntaxe.y | Somm

```
1   #include<stdio.h>
2   #include<stdlib.h>
3   #include<string.h>
4    /****    fonction main ********* */
5   void main(){
6   int A,B,S;
7   A=2;
8   B=3;
9   S=A+B;
10      printf("%d",S);
11  /****        La    FIN        *** */
12  }
```

```
C:\Users\kawtar\Desktop\Compilation\td_Generation_code>flex -olexiqueL.c lexique.l

C:\Users\kawtar\Desktop\Compilation\td_Generation_code>bison -d -osyntaxeY.c syntaxe.y

C:\Users\kawtar\Desktop\Compilation\td_Generation_code>gcc -o prog lexiqueL.c syntaxeY.c

C:\Users\kawtar\Desktop\Compilation\td_Generation_code>prog.exe  0<some.txt
Expression correct

C:\Users\kawtar\Desktop\Compilation\td_Generation_code>gcc somme.c -o somme

C:\Users\kawtar\Desktop\Compilation\td_Generation_code>somme
5
C:\Users\kawtar\Desktop\Compilation\td_Generation_code>
```

Le Code intermédiaire fichier.c

Le résultat donne 5

Thank you!