République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Mohamed Khider Biskra

Faculté des Sciences Exactes et des Sciences de la Nature et de la Vie
Département d'informatique

Polycopié : Compilation
Pour 3ème Année Licence
Spécialité : Systèmes informatique

Par

MEADI Mohamed Nadjib, Maitre de conférences B

Année Universitaire 2022/2023

Table Des Matières

Liste des figures

Chapitre 01 : Introduction

1 Objectif

La compilation est une transformation que l'on fait subir à un programme écrit dans un langage évoluée pour le rendre exécutable. Fondamentalement, c'est une traduction : un texte écrit en Pascal, 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.

Une autre phase importante qui intervient après la compilation pour obtenir un exécutable est la phase d'éditions de liens. Un éditeur de liens résout entre autres les références à des appels de routines dont le code est conservé dans des librairies. En général, un compilateur comprend une partie éditeur de liens.

La "compilation" n'est pas limitée à la traduction d'un programme informatique écrit dans un langage de haut niveau en un programme directement exécutable par une machine, cela peut aussi être : la traduction d'un langage de programmation de haut niveau vers un autre langage de programmation de haut niveau. Par exemple une traduction de Pascal vers C, ou de Java vers C++,... etc. On parle plutôt de **traducteur** c-à-d La traduction d'un langage quelconque vers un autre langage quelconque (ie pas forcement de programmation) : word vers html, pdf vers ps, ...

Le but de ce cours est de présenter les principes de base inhérents à la réalisation de compilateurs. Les idées et techniques développées dans ce domaine sont si générales et fondamentales qu'un informaticien les utilisera très souvent au cours de sa carrière : traitement de données, moteurs de recherche, etc. ..

Comprendre comment est écrit un compilateur permet de mieux comprendre les "contraintes" imposées par les différents langages lorsque l'on écrit un programme dans un langage de haut niveau.

2 Structure d'un compilateur

La compilation se décompose en deux phases :

- Une phase d'analyses, qui va reconnaître les variables, les instructions, les opérateurs et élaborer la structure syntaxique du programme ainsi que certaines propriétés sémantiques
- Une phase de synthèse et de production qui devra produire le code cible:

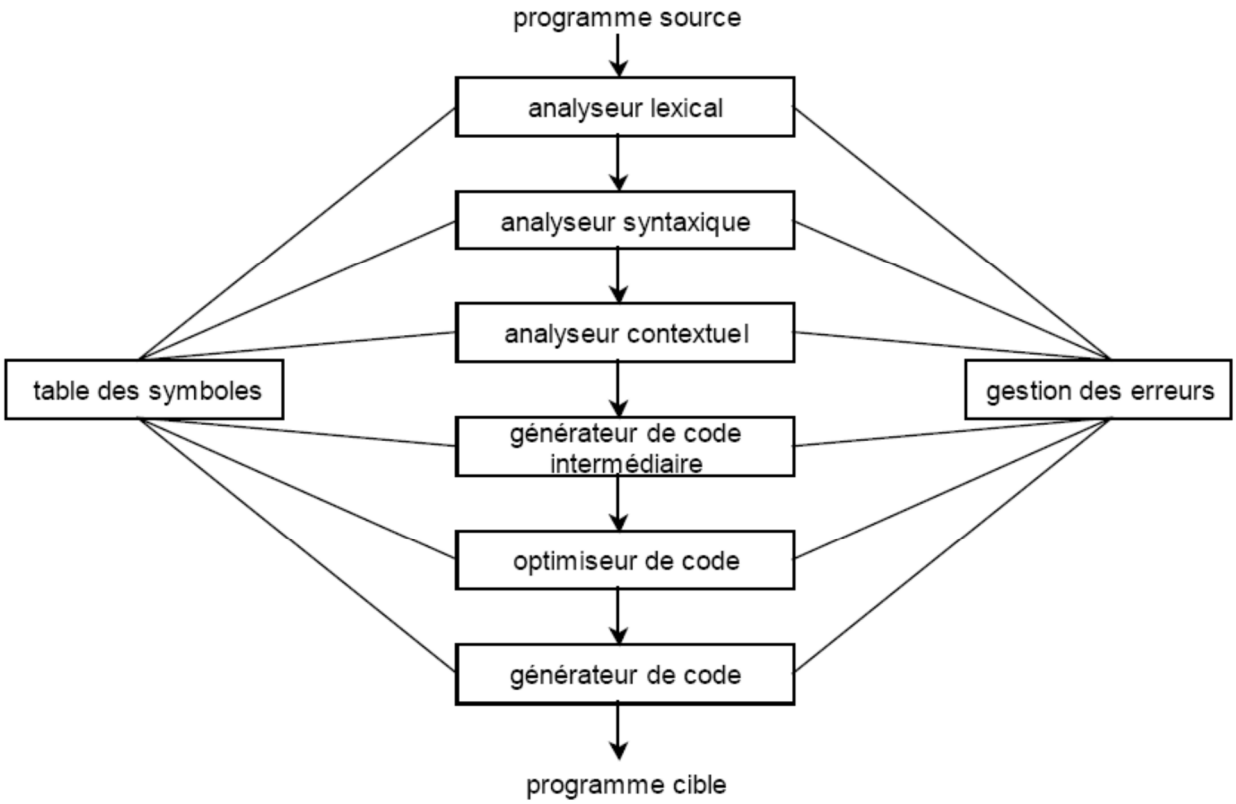


Figure 1.1: Les différentes phases composants un compilateur

2.1 Une phase d'analyses

1. **Analyse lexicale** (appelée aussi Analyse linéaire)

L'objectif de cette étape est de reconnaître les "types" des "mots" lus par le compilateur. La lecture du programme source se fait de gauche à droite et les caractères lus seront regroupés en unités lexicales.

L'analyse lexicale se charge de

- L'élimnation des caractères superflus (commentaires, espaces, ...)
- identifier les parties du texte qui ne font pas partie à proprement parler du programme mais sont des directives pour le compilateur
- identifier les symboles qui représentent des identificateurs, des constantes réelles, entière, chaînes de caractères, des opérateurs (affectation, addition, ...), des séparateurs (parenthèses, points virgules, ...), les mots clefs du langage, ... C'est cela que l'on appelle des unités lexicales.

Outils théoriques utilisés : expressions régulières et automates à états finis

Exemple

Reprenons l'exemple de l'affectation : temps := sec + minute * 60.

La phase d'analyse lexicale va lire la séquence de caractères correspondant à l'affectation. Elle va éliminer les marques de ponctuation et les séparateurs de mots. Ainsi, elle va nous donner dans l'ordre la liste d'unités lexicales suivante :

1. L'identificateur temps ;
2. Le symbole d'affectation := ;
3. L'identificateur sec ;
4. Le signe plus (+) ;
5. L'identificateur minute ;
6. Le signe de multiplication (*) ;
7. Le nombre 60.

Le texte source comporte donc sept unités lexicales, qu'on appelle aussi lexèmes.

2. **Analyse syntaxique** (appelée aussi Analyse hiérarchique ou Analyse grammaticale)

Il s'agit de regrouper les unités lexicales en structures grammaticales, de découvrir la structure du programme. L'analyseur syntaxique sait comment doivent être construites les expressions, les instructions, les déclarations de variables, les appels de fonctions, ...

Outils théoriques utilisés : grammaires et automates à pile.

3. **Analyse sémantique** (appelée aussi analyse contextuelle)

Dans cette phase, on opère certains contrôles (contrôles de type, par exemple) afin de vérifier que l'assemblage des constituants du programme a un sens. Outil théorique utilisé : schéma de traduction dirigée par la syntaxe

- les identificateurs apparaissant dans les expressions ont-ils été déclarés ?
- les opérandes ont-ils les types requis par les opérateurs ?
- les opérandes sont-ils compatibles ? n'y a-t-il pas des conversions à insérer ?
- les arguments des appels de fonctions ont-ils le nombre et le type requis ?
- etc.

2.2 Une phase de synthèse

4. **Génération du code intermédiaire** Il s'agit de produire les instructions en langage cible.

En règle générale, le programmeur dispose d'un calculateur concret (cartes équipées de processeurs, puces de mémoire, ...). Le langage cible est dans ce cas défini par le type de processeur utilisé.

Mais si l'on écrit un compilateur pour un processeur donné, il n'est alors pas évident de porter ce compilateur (ce programme) sur une autre machine cible.

Pour cela, on introduit des machines dites abstraites qui font l'abstraction des architectures réelles existantes. Ainsi, on s'attache plus aux principes de traduction, aux concepts des langages, qu'à l'architecture des machines.
En général, on produira dans un premier temps des instructions pour une machine abstraite (virtuelle). Puis ensuite on fera la traduction de ces instructions en des instructions exécutables par la machine réelle sur laquelle on veut que le compilateur s'exécute. Ainsi, le portage du compilateur sera facilité, car la traduction en code cible virtuel sera faite une fois pour toutes, indépendamment de la machine cible réelle. Il ne reste plus ensuite qu'à étudier les problèmes spécifiques à la machine cible, et non plus les problèmes de reconnaissance du programme (cf Java).

5. **Optimisation du code** Cette phase tente d'améliorer le code produit de telle sorte que le programme résultant soit plus rapide. Par Exemple
    - détecter l'inutilité de recalculer des expressions dont la valeur est déjà connue,
    - transporter à l'extérieur des boucles des expressions et sous-expressions dont les opérandes ont la même valeur à toutes les itérations
    - détecter, et supprimer, les expressions inutiles
    - etc.
6. **Génération du code machine.** Cette phase nécessite la connaissance de la machine cible (réelle, virtuelle ou abstraite), et notamment de ses possibilités en matière de registres, piles, etc.

2.3 Phase parallèle

7. **Gestion de la table des symboles** La table des symboles est la structure de données utilisée pour stocker les informations qui concernent les identificateurs du programme source (par exemple leurs types, leurs emplacements dans la mémoire, leurs portées, visibilités, nombre et type et mode de passage des paramètres d'une fonction, ..etc.
   Le remplissage de cette table a lieu lors des phases d'analyse. Les informations contenues dans la table des symboles sont nécessaires lors des analyses syntaxique et sémantique, ainsi que lors de la génération de code.

8. **Gestion des erreurs** Chaque phase peut rencontrer des erreurs. Cependant, après avoir détecté une erreur, une phase doit la traiter de telle façon que la compilation puisse continuer et que d'autres erreurs dans le programme source puissent être détectées. Un compilateur qui s'arrête à la première erreur n'est pas non plus très performant. Bien sûr, il y a des limites à ne pas dépasser et certaines erreurs peuvent entrainer l'arrêt de l'exécution du compilateur.

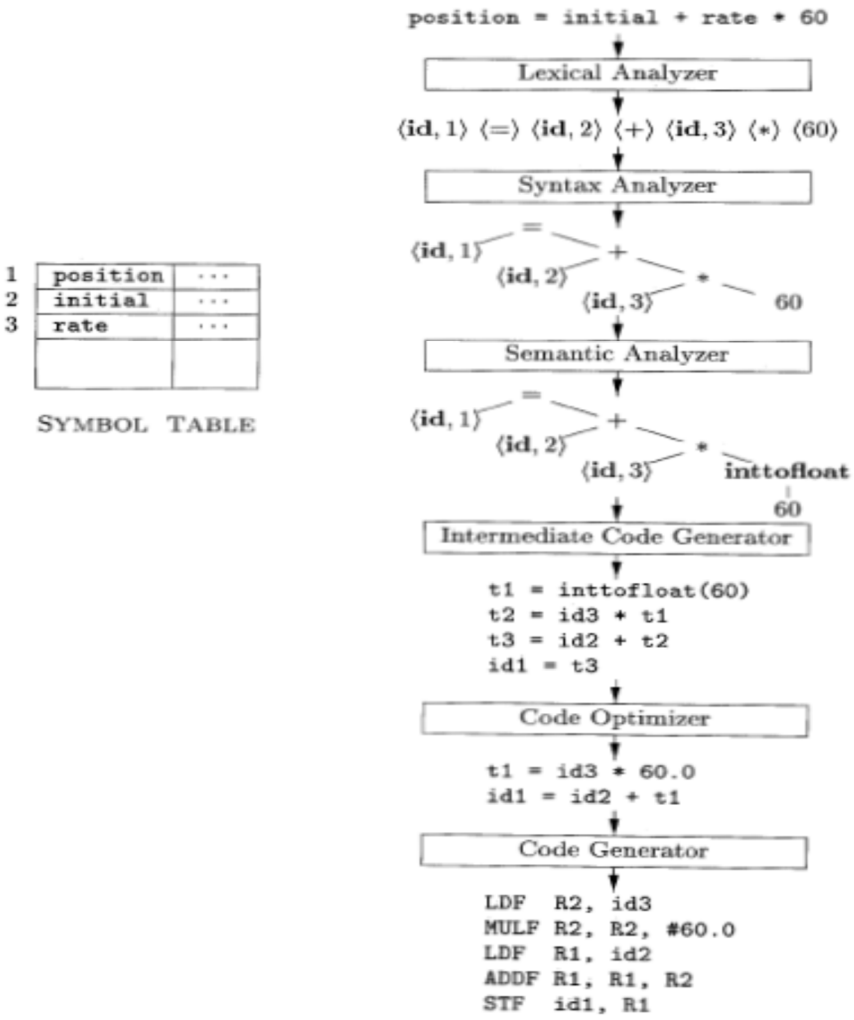


Figure 1.2: Exemples de différentes étapes composant un compilateur

Chapitre 02 : Analyse lexicale

1 Introduction

L'analyseur lexical constitue la première étape d'un compilateur. Sa tâche principale est de lire les caractères d'entrée et de produire comme résultat une suite d'unités lexicales que l'analyseur syntaxique aura à traiter.

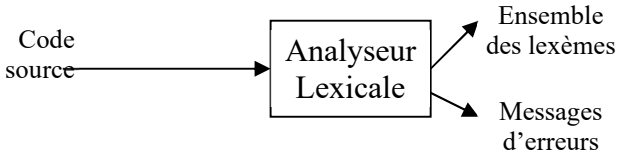


Figure 2.1 : Les entées et les sorties d'un analyseur lexicale

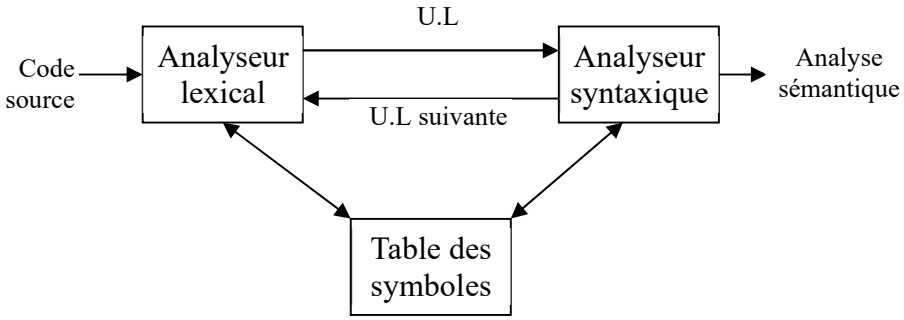


Figure 2.2 : La coopération entre l'analyseur lexicale et l'analyseur syntaxique

En plus, l'analyseur lexical réalise certaines tâches secondaires comme par Exemple

1) L'élimination de caractères superflus (commentaires, tabulations, fin de lignes, ...),
2) Gère aussi les numéros de ligne dans le programme source pour pouvoir associer à chaque erreur rencontrée par la suite la ligne dans laquelle elle intervient.

2 Définitions

1. **Une unité lexicale** est une suite de caractères qui a une signification collective.
2. **Un lexème** toute suite de caractère du programme source qui concorde avec **le modèle** d'une unité lexicale.
3. **Un modèle (Règle lexicale**) est une règle associée à une unité lexicale qui décrit l'ensemble des chaînes du programme qui peuvent correspondre à cette unité lexicale.
4. **Attributs** : informations concernant le lexème (champs dans la table des symboles) ;

Exemples

Unité lexicale	Modèle	Lexème
opérateurs relationnels	>,>, =, >=, >=.	>,>, =, >=, >=
ID (identificateurs)	Suite de caractères composée de chiffres, lettres et qui commencent par une lettre	truc, i, x1, ajouter_valeur
NOMBRE	Suite non vide de chiffres	12, 83204, 0
REEL	Lexème correspondant à l'unité lexicale NOMBRE suivi éventuellement d'un point et d'une suite de chiffres, le tout suivi éventuellement du caractère *E* (ou *e*)	12.4, 0.5e3, 10., -4e-1, -.103e+2

	et d'un lexème correspondant à l'unité lexicale NOMBRE. Cela peut également être un point suivi d'une suite de chiffres, et éventuellement du caractère *E* (ou *e*) et d'un lexème correspondant à l'unité lexicale NOMBRE.	
if	le caractère *i* suivi par *f*	if
Const	Const	Const
Chaîne	toute suite de caractères (sauf “) mis entre ” et “	“Bonjour”

3 Expressions régulières

Une expression régulière est une notation pour décrire un langage régulier.
Soit A un alphabet (un ensemble de lettres), une expression régulière est donc:

1. Les éléments de A, ε et ∅ sont des expressions régulières.
2. Si α et β sont des expressions régulières, alors (α | β), (αβ) et α* sont des expressions régulières. (α | β) représente l'union, (αβ) la concaténation et α* la répétition (un nombre quelconque de fois). ε est l'élément neutre par rapport à la concaténation et ∅ est l'ensemble vide de caractère, neutre par rapport à l'union.

Exemples : On décrit les lexèmes par des expressions régulières.
Identificateur = alpha (alpha | numer)*
Les mots clés: "for", "if"
Les variables: ['a'-'z']+ ['0'-'9']*
Les entiers: ['0'-'9']+
Les symboles: '(', ')', '+', '*', '='
Le lexème vide: (' ' | '\n')

L'ordre de priorité

Les opérateurs ∗, concaténation et | sont **associatifs à gauche**, et vérifient

1. ∗
2. concaténation
3. |

4 Définition régulière

La nomination des expressions régulières est dite une définition régulière. Ces noms seront utilisés pour construire d'autres expressions régulières. On écrit donc

$d_1 \rightarrow r_1$
$d_2 \rightarrow r_2$
. . .
$d_n \rightarrow r_n$

où chaque d_i est un nom distinct et chaque r_i est une expression régulière sur l'alphabet A U $\{d_1,d_2,d_{i-1}\}$

Exemple

Voici quelques définitions régulières définissantles identificateurs et les nombres du langage Pascal :

lettre → A | B | . . . | Z | a | b | . . . | z
chiffre → 0 | 1 | . . . | 9
identificateur → lettre (lettre | chiffre)∗

chiffres → chiffre chiffre *
frac → . chiffres | ε
Exp → (E (+ | - | ε) chiffres) | ε
nombre → chiffres frac exp

Exemple

5 Automates

Un automate à états finis (AEF) est défini par

- Un ensemble fini E d'états
- Un état e_0 distingué comme étant l'état initial
- Un ensemble fini T d'états distingués comme états finaux (ou états terminaux)
- Un alphabet A des symboles d'entrée
- Une fonction de transition □:A × E → E qui à tout couple formé d'un état et d'un symbole de fait correspondre un ensemble (éventuellement vide) d'états : $\delta (e_i,a)=\{ei1,\ldots,e_{in}\}$

Les automates sont souvent donnés sous la forme d'un graphe: les états sont les nœuds du graphe et les arcs correspondent à la fonction de transition.

Le langage reconnu par un automate est l'ensemble des chaînes qui permettent de passer de l'état initial à un état terminal.

Exemple

A={a,b}, E={0,1,2,3}, e_0=0, T={3}

δ(0,a)={0,1}, δ(0,b)={0}, δ(1,b)={2}, δ(2,b)={3},

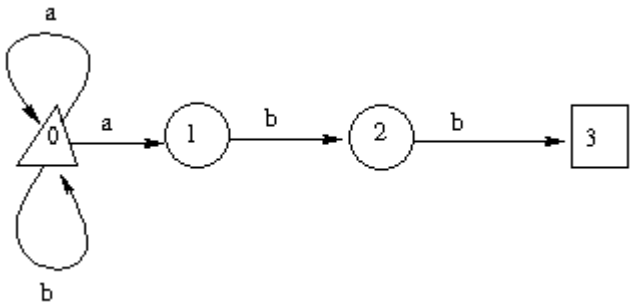
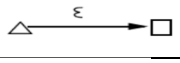
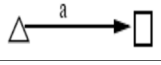
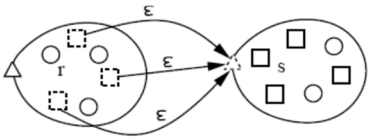
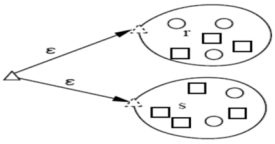
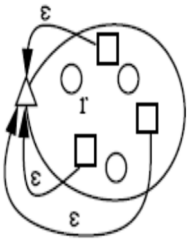


Figure 2.3: Représentation graphique d'un automate

5.1 Construction d'un AFN à partir d'une E.R.

Pour une expression régulière S, on note A(s) un automate reconnaissant cette expression;

Expression régulière (S)	**Automate équivalent A(s)**
Automate acceptant la chaîne vide	
Automate acceptant la lettre a	
Automate acceptant (r)(s) 1. mettre une ε -transition de chaque état terminal de A(r) vers l'état initial de A(s) 2. les états terminaux de A(r) ne sont plus terminaux 3. le nouvel état initial est celui de A(r) 4. (l'ancien état initial de A(s) n'est plus état initial)	
Automate reconnaissant r|s 1. créer un nouvel état initial q 2. mettre une ε-transition de q vers les états initiaux de A(r) et A(s) (les états initiaux de A(r) et A(s) ne sont plus états initiaux)	
Automate reconnaissant r+	

5.2 Définition d'un automates finis déterministes (AFD)

Un automate fini est dit déterministe lorsqu'il ne possède pas de ε-transition et lorsque pour chaque état e et pour chaque symbole a, il y a au plus un arc étiqueté a qui quitte e . On appelle ε-transition, une transition par le symbole ε entre deux états, Considérer des ensembles d'états plutôt que des états.

Il existe des algorithmes permettant de déterminiser un automate non déterministe (c'est à dire de construire un AFD qui reconnait le même langage que l'AFN donné). L'AFD obtenu comporte en général plus d'états que l'AFN, donc le programme le simulant occupe plus de mémoire.

6 Construction d'un analyseur lexical

Généralement, Il existe deux méthodes pour construire un analyseur lexical : **manuellement** ou en utilisant des outils d'aides comme Lex par exemple.
Une méthode simple d'implantation d'un analyseur lexical consiste à construire un automate (diagramme) traduisant la structure des unités lexicales et ensuite à traduire à la main cet automate en code.

Exemple

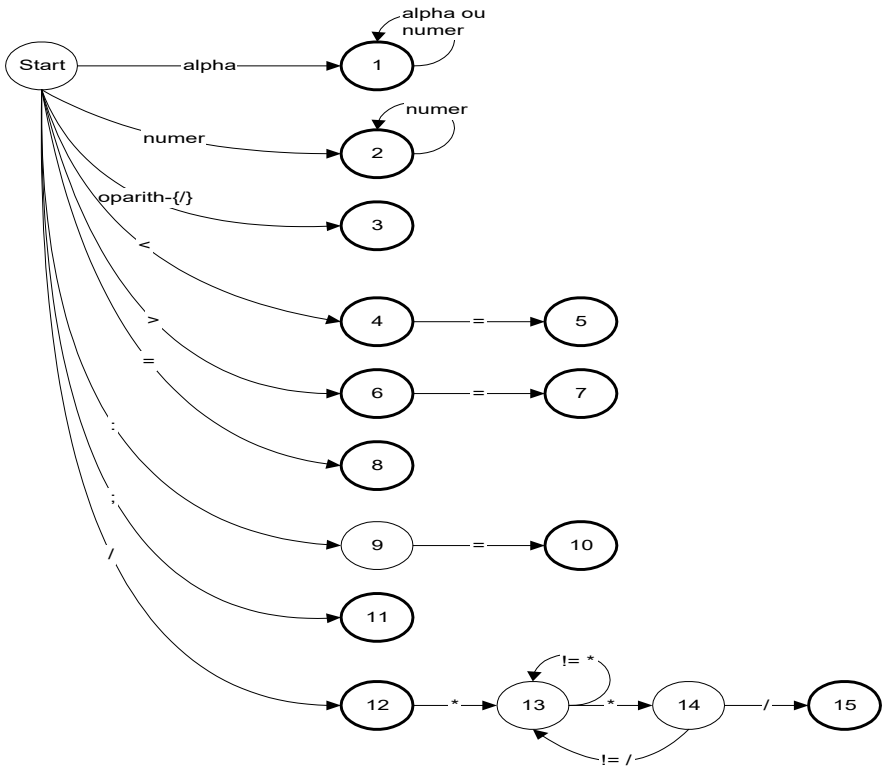


Figure 2.4 : Exemple d'un automate spécifiant quelques opérateurs arithmétiques et logiques

Le code implémentant l'automate de la Figure 2.7 est comme suit :

```
buffer=&Buffer[0];
while(1) {
    c = lireChar();
    switch(etat){
        case: 0
        switch(type(c)){
        buffer*=c;buffer++;
        case ALPHA: etat=1;break;
        case NUMER: etat=2;break;
        case OPARITH-{/}: etat=3; break;
        case '<': etat=4; break;
        case '>': etat=6; break;
        case '=': etat=8; break;
        case ':': etat=9; break;
        case ';': etat=11; break;
        case '/': etat=12; break;
        default: erreur;
        }
        break;
        case 1:
        if (type(c) == ALPHA ou type(c) == NUMER) {
        buffer*=c;buffer++;
        etat=1;
        }
        else {
```

```
        buffer*='\0'; remettreCar();
        return(ajouter_token(Buffer,TYPE_IDENT));
        }
        break;
        case 2:
        if (type(c) == NUMER){
        buffer*=c;buffer++;
        etat=2;
        }
        else {
        buffer*='\0';remettreCar();
        return(ajouter_token(Buffer,TYPE_NUMER));
        }
        break;
        case 3:
        return(ajouter_token(Buffer,TYPE_OPER)); break;
        case 4:
        if (c == '=') {
        buffer*=c;buffer++;
        etat = 5;
        } else {
        remettreCar();
        return(est-dans("<"));
        }
        break;
{{à finir!!}}
        }
```

7 Outil Lex (générateur des analyseurs lexicaux)

Lex (fLex) est un produit Gnu. Lex accepte en entrée des spécifications d'unités lexicales sous forme de définitions régulières et produit un programme écrit dans le langage C qui,

Une fois compilé à l' ide de la commande :

```
> Lex fichier.l
```

donne le fichier Lex.yy.c qu'il faut compiler avec un compilateur C, comme par Exemple

```
> gcc Lex.yy.c –l l
```

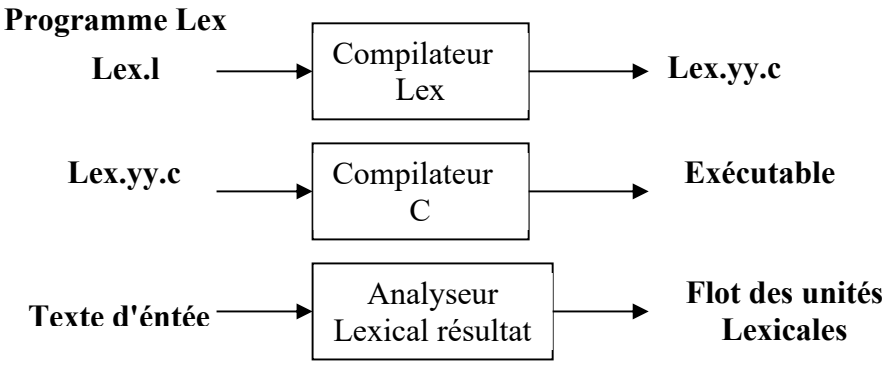


Figure 2.5 : Les étapes à suivre pour obtenir un analyseur lexical à partir d'un fichier Les

7.1 Les expressions régulières Lex

Une expression régulière Lex se compose de caractères normaux et de méta-caractères, qui ont une signification spéciale : $, \, ^, [,], {,}, <, >, +, -, *, /,|,?. Le tableau suivant donne les expressions régulières reconnues par Lex. Attention, Lex fait la différence entre les minuscules et les majuscules.

Expression	Signification	Exemple
c	tout caractère *c* qui n'est pas opérateur ou métacaractère	a
\c	caractère littéral c (lorsque c est un métacaractère)	\+ \.
"s "	chaîne de caractères	"bonjour "
.	n'importe quel caractère, sauf retour à la ligne (\n)	a.b
^	l'expression qui suit ce symbole débute une ligne	^abc
$	l'expression qui précède ce symbole termine une ligne	abc$
[s]	n'importe quel caractère de s	[abc]
[^s]	n'importe quel caractère qui n'est pas dans s	[^xyz]
r*	0, 1 ou plusieurs occurrences de r	b*
r+	1 ou plusieurs occurrences de r	a+
r?	0 ou 1 occurrence de r	d?
r{m}	m occurrences de r	e{3}
r{m,n}	entre m et n occurrences de r	f{2,4}
r1r2	r1 suivie de r2	ab
r1|r2	r1 ou r2	c|d
r1/r2	r1 si elle est suivie de r2	ab/cd
(r)	r	(a|b)?c
<x>r	r si Lex se trouve dans l'état x	<x>abc

7.2 Structure d'un fichier Lex

Le fichier de spécifications Lex contient des expressions régulières suivies d'actions (règles de traduction). L'analyseur Lexical obtenu lit le texte d'entrée caractère par caractère jusqu'à ce qu'il trouve le plus long préfixe du texte d'entrée qui corresponde à l'une des expressions régulières. Dans le cas où plusieurs règles sont possibles, c'est la première règle rencontrée (de haut en bas) qui l'emporte. Il exécute alors l'action correspondante. Dans le cas où aucune règle ne peut être sélectionnée, l'action par défaut consiste à copier le caractère du fichier d'entrée en sortie.

Un fichier de description pour Lex est formé de trois parties, selon le schéma suivant :

```
%{
Déclaration en C des variables, des constants,...etc.
%}
Déclaration des définitions régulières.
%%
Expression régulières et actions correspondantes
%%
Déclaration des procédures auxiliaires
```

Dans lequel aucune partie n'est obligatoire. Cependant, le `séparateur "%%"` est utilisé pour séparer entre les déclarations et expression régulières et actions correspondantes (le productions).

7.2.1 Partie des déclarations

La section déclarations peut elle même se composer de :
- un bloc littéral
- des définitions
- des conditions de départ

a. Bloc littéral

Cette partie d'un fichier Lex peut contenir :
- Commence par **%{** et se termine par **%}**. Où **%{** et **%}** doivent être placés en début de ligne.
- Contient des déclarations et définitions en C.
- Est copié tel quel dans le fichier Lex.yy.c produit par la commande Lex
- Les définitions et déclarations qu'il contient sont globales au programme produit par Lex.

b. Définitions

Associations d'identificateurs à des expressions régulières. . Ces définitions sont de la forme :

notion expression régulière

Exemple
separ[\t\n]
espace {separ}$^{+}$
lettre [A-Za-z]
chiffre [0-9]
ident{lettre}({lettre}|{chiffre})*
nbre{chiffre}$^{+}$(\.{chiffre}$^{+}$)?(E[+\-]?{chiffre}$^{+}$)?

Remarque :

Lorsqu'on se réfère à une expression régulière en utilisant un identificateur, celui-ci est mis entre accolades.

c. Conditions de départ

Les conditions de départ permettent de définir plusieurs états de Lex

Exemple

%start etat1 etat2 ….
Où etat1, état2 … sont les états possibles de Lex

7.2.2 Partie des productions

Contient deux parties

Partie gauche :

- spécification des expressions régulières reconnues
- pour une chaîne de lettres et de chiffres, les guillemets peuvent être omis
- identificateurs d'expressions mis entre accolades

Partie droite :

- actions exécutées lorsque unités Lexicales reconnues
- actions définies avec syntaxe C
- si scanner appelé par parser YACC, alors :

- les attributs de l'unité Lexicale reconnue doivent être déposés dans **yylval**
- l'unité Lexicale reconnue doit être retournée

Exemple

```
{ \t\n}{ /* pas d 'action */ }
``<=`` {return(PPE);}
``<>`` {return(DIF);}
``<`` {return(PPQ);}
``=`` {return(EGA);}
``>=`` {return(PGE);}
``>`` {return(PGQ);}
``:=`` {return(AFF);}
 if{return(si);}
 then{return(alors);}
else{return(sinon);}
{ident}{yylval = yytext; return(id);}
{nbre}{yylval = yytext; return(nb);}
```

7.2.3 Section Procédures auxiliaires

Section optionnelle qui permet de :
- définir toutes les fonctions utilisées dans les actions associées aux expressions reconnues
- définir (si nécessaire) le programme principal (main()).

Exemple

Cet exemple compte le nombre de voyelles, consonnes et caractères de ponctuations d'un texte entré au clavier.

```
%{
  int nbVoyelles, nbConsonnes, nbPonct;
%}
consonne      [b-df-hj-np-tv-xz]
ponctuation   [,;:?!\.]
%%
[aeiouy]        nbVoyelles++;
{consonne}      nbConsonnes++;
{ponctuation}   nbPonct++;
.|\n            // ne rien faire
%%
main(){
nbVoyelles = nbConsonnes = nbPonct = 0;   yylex();
printf("Il y a %d voyelles, %d consonnes et %d ponctuations.\n",nbVoyelles, nbConsonnes,
nbPonct); }
```

7.3 Variables et fonctions prédéfinies

char yytext[ ] : tableau de caractères qui contient la chaîne d'entrée qui a été acceptée.
int yyleng : longueur de cette chaîne.
int yylex() : fonction qui lance l'analyseur (et appelle `yywrap()`).
yylval: retourne la valeur associé à l'unité lexicale reconnue**.**
ECHO afficher l'unité lexicale reconnue (équivalente à **printf("%s",yytext);**)
FILE *yyout: fichier de sortie.
FILE *yyin: fichier d'entrée

int yywrap(): fonction toujours appelée en fin de flot d'entrée. Elle ne fait rien par défaut, mais l'utilisateur peut la redéfinir dans la section des fonctions auxiliaires. **`yywrap()`** retourne 0 si l'analyse doit se poursuivre (sur un autre fichier d'entrée) et 1 sinon.
unput(char c) : remet le caractère dans le flot d'entrée.
int yylineno : numéro de la ligne courante.
yymore(): fonction qui concatène la chaîne actuelle **yytext** avec celle qui a été reconnue avant
yyless(): fonction admettant un entier comme argument, yyless(k>0) :

- supprime les **(yyleng-k)** derniers caractères de yytext, dont la longueur devient alors k
- recule le pointeur de lecture sur le fichier d'entrée de **(yyleng-k)** positions, les caractères supprimés de **yytext** seront donc considérés pour la reconnaissance des prochaines unités

yyterminate() : fonction qui stoppe l'analyseur .

Exemple

Cet exemple insert le numéro de ligne à chaque ligne dans un fichier. Certaines implémentations de lex prédéfinir et de calculer *le nombre des lignes*. Le fichier d'entrée est est yyin, et par défaut l'entrée standard.

```
%{
int yylineno;
%}
%%
^(.*)\n printf("%4d\t%s", ++yylineno, yytext);
%%
int main(int argc, char *argv[]) {
yyin = fopen(argv[1], "r");
yylex();
fclose(yyin);
}
```

8 Erreurs Lexicales

Peu d'erreurs sont détectables au seul niveau Lexical :

Plusieurs stratégies sont possibles :

- mode panique : on ignore les caractères qui posent problème et on continue. Cette technique se contente de refiler le problème à l'analyseur syntaxique

- transformations du texte source : insérer un caractère, remplacer, échanger, etc. Elle se fait en calculant le nombre minimum de transformations à apporter au mot qui pose problème pour en obtenir un qui ne pose plus de problèmes. Cette technique de récupération d'erreur est très peu utilisée en pratique car elle est trop coûteuse à implanter.

9 Exercices

Exercice 01

Soit l'alphabet A = {0,1}. Définir les expressions régulières pour les langages ci-dessous :
1. Toutes les chaînes qui se terminent par 00.
2. Toutes les chaînes dont le 10ème symbole, compté à partir de la fin de la chaîne, est un 1.
3. Ensemble de toutes les chaînes dans lesquelles chaque paire de 0 apparaît devant une paire de 1.
4. Ensemble de toutes les chaînes ne contenant pas 101.

Exercice 02

Définir les expressions régulières pour les langages ci-dessous :
1. Le langage contenant tous les mots **begin** écrits avec des majuscules et minuscules mélangés : bEgin, BEGin, begiN, BEGIN, …
2. Les nombres octaux sans signe (ils commencent toujours par un 0 et ne contiennent que des chiffres compris entre 0 et 7).
3. Les nombres flottants : .45, 2., -4.5, .45e23, 2.E-4, 5.34e+2,… On n'acceptera pas un point tout seul avec ou sans la partie exponentiation : ., .e23, …
4. Les mots composés de lettres en minuscule de l'alphabet et contenant exactement les chiffres 1, 2, 3 et 4 toujours présents dans l'ordre croissant : 1234, 1ab2c34, abc12f3ad4, …
5. A la place des chiffres 1, 2, 3 et 4, on choisit à la place les lettres a, b, c et d.
6. Les mots non vides, composés de lettres minuscules de l'alphabet, mots qui ne commencent jamais par la lettre 'c',
7. Les mots non vides commençant par une lettre minuscule de l'alphabet, et chaque lettre est toujours suivie d'un chiffre. Ces mots sont de longueur paire :
   e1, b2c1d3, ... sont des mots acceptés,
   e, 1, da1d, abcd, ... ne sont pas acceptés.
8. les mots non vides, composés de caractères minuscules de l'alphabet, mots où le caractère 'e' est soit complètement absent ou soit présent toujours par paire :
   abc, ee, beecee, eeeedaf, ... sont des mots acceptés,
   e, bec, deeea, ... ne sont pas acceptés.
9. Les mots non vides
   – composés de lettres minuscules de l'alphabet,
   – si 'a' apparaît 1 ou plusieurs fois, il n'y aura pas de 'b',
   – si 'b' apparaît 1 ou plusieurs fois, il n'y aura pas de 'a'.
10. Les mots non vides, composés de lettres minuscules ou majuscules de l'alphabet, ne contenant au plus qu'un seul 'e' ou 'E' à la fois :
    E, ea, aabCDfg, ... sont des mots acceptés,
    aebE, ab12de, abcdEfeF, abecEdae, ... ne sont pas acceptés.
11. Les mots composés de lettres minuscules de l'alphabet,
    – où 'a', 'b' et 'c' apparaissent au plus une fois (c'est-à-dire 0 ou 1 fois),
    – si 'a' est présent, il n'y aura pas 'b' ni 'c' à sa gauche de près ou de loin,
    – si 'b' est présent, il n'y aura pas 'c' à sa gauche de près ou de loin.
    Par Exemple a, b, c, abde, btgcf, xaxbefc, ... sont des mots acceptés, aeaf, ebdag, acbde, ... ne sont pas acceptés.

Exercice 03

1. Donner une expression régulière étendue permettant de reconnaître des entiers naturels :
   - sans zéro non significatif à gauche,
   - composés de nombre pair de chiffres pairs,
   - ou composés de nombre impair de chiffres impairs.

   Exemples d'entiers acceptés: 224466 246820 135 9975311

Exemples d'entiers non acceptés: 024666 2410 1357 13257

2. Donner une expression rationnelle étendue permettant de reconnaître des durées (strictement inférieures à 24h) en heures, minutes et secondes respectant les contraintes suivantes :
   - Une durée s'exprime avec le format `--h--m--s`.
   - Les secondes et minutes sont comprises entre 0 et 59 et composées d'un ou deux chiffres.
   - Les heures sont comprises entre 0 et 23 et composées d'un ou deux chiffres.
   - Les heures ne peuvent être présentes que si les minutes y sont et les minutes ne peuvent être présentes que si les secondes y sont.

   Exemples de durées acceptées : 05s 05m00s 02h3m04s 2h03m4s
   Exemples de durées non acceptées: 05m : sans la présence des secondes
3. Donner une expression rationnelle étendue en Lex permettant de reconnaître les numéros d'immatriculation composés de :
   - 3 à 5 chiffres ne commençant par le chiffre 0,
   - suivis de 2 lettres majuscules en excluant WW,
   - terminant par 2 chiffres en excluant 00

   Exemples de numéros acceptés :123WA01 64725AA45

Exercice 4

1. Donnez l'expression régulière étendue (ERE) qui désigne n'importe quelle suite de 5 caractères, y compris \n.
2. Donnez l'ERE qui désigne une chaîne formée de n'importe quel nombre de \, suivi de n'importe quel nombre de *.
3. Les shells UNIX permettent d'écrire des fichiers batch dans lesquels on peut insérer des commentaires. Une ligne est considérée comme commentaire si elle commence par #. Quelle est l'ERE qui accepte de tels commentaires ?
4. Donnez l'ERE acceptant l'ensemble des phrases «correctes» selon les critères suivants :
   - Le premier mot de la phrase a une majuscule ;
   - la phrase se termine par un point ;
   - la phrase est composée d'un ou plusieurs mots (a...z et A...Z), séparés par un espace ;
   - on trouve une phrase par ligne.

   Remarquons que les caractères de ponctuation autres que le point ne sont pas admis.
5. Écrivez l'ERE qui accepte tous les noms de fichiers DOS (composés de 8 caractères : A...Z, a...z et _), dont l'extension est ext et commençant par la chaîne abcde. Attention, l'ERE ne doit accepter que le nom du fichier sans l'extension !

Exercice 5

Ecrire une fonction d'analyse lexicale à l'aide de LEX. Les *tokens* reconnus seront :

- Les nombres décimaux (en notation scientifique) ;
- Les identifiants de variables ;
- Les opérateurs relationnels;
- Les mots-clefs si, sinon et alors.

Cette fonction a pour but d'être utilisée dans un analyseur syntaxique écrit en YACC.

Exercice 6

Donner un fichier d'entrée pour Lex permettant

1. de lire, jusqu'à la fin de la source d'entrée, les nombres entiers positifs écrits en octal, en décimal ou en hexadécimal, les nombres flottants,
2. de les afficher avec une indication du type du nombre lu (octal, décimal, hexadécimal ou flottant),
3. de compter le nombre de ligne tapé par l'utilisateur,
4. d'indiquer les commentaires C, sans oublier de compter le nombre de ligne (on propose d'utiliser les conditions), et afficher à la fin le nombre total de lignes.

Chapitre 03 : Analyse syntaxique

1 Introduction

Tout langage de programmation possède des règles qui indiquent la structure syntaxique d'un programme bien formé. Par exemple, en Pascal, un programme bien formé est composé de blocs, un bloc est formé d'instructions, une instruction de ... . La **syntaxe** d'un langage peut être décrite par une **grammaire**.

L'analyseur syntaxique reçoit une suite d'unités lexicales de la part de l'analyseur lexical et doit vérifier que cette suite peut être engendrée par la grammaire du langage. Le problème est donc :

- Etant donnée une grammaire G.
- Etant donnée un mot *m (un programme)*

➔Est-ce que *m* appartient au langage généré par *G* ?

Le principe est d'essayer de construire un arbre de dérivation. Il existe deux méthodes pour cette construction : méthode (analyse) descendante et méthode (analyse) ascendante.

2 Grammaires hors contexte

Pour décrire la syntaxe d'un langage de programmation, on utilise une grammaire. Une grammaire est un ensemble de règles décrivant comment former des phrases.

Définition 1 Une **grammaire** est la donnée de $G=(V_T,V_N,S,P)$ où

- V_T est un ensemble non vide de symboles ***terminaux*** (alphabet terminal)
- V_N est un ensemble de symboles ***non-terminaux***, avec $\mathbf{V_T} \cap \mathbf{V_N}$
- S est un symbole initial $\in \mathbf{V_N}$ appelé ***axiome***
- P est un ensemble de ***règle de productions***.

Définition 2

Une ***règle de production*** $\alpha \rightarrow \beta$ précise que la séquence de symboles α $(\alpha \in (\mathbf{V_T} \mathbf{U} \mathbf{V_N})^+)$ peut être remplacée par la séquence de symboles β $(\beta \in (\mathbf{V_T}\ \mathbf{U}\ \mathbf{V_N})^*)$. α est appelée ***partie gauche*** de la production, et β ***partie droite*** de la production.

Exemple

Symboles terminaux (alphabet) : $\mathbf{V_T=\{a,b\}}$
Symboles non-terminaux :$\mathbf{V_N=\{S\}}$
Axiome : *S*
Règles de production $\begin{cases} \mathbf{S} \rightarrow \varepsilon \\ \mathbf{S} \rightarrow \mathbf{aSb} \end{cases}$ qui se résument en $\mathbf{S} \rightarrow \varepsilon \mid \mathbf{aSb}$

Considérons la grammaire suivante :
Expr → Expr Op nb| nb
Op → +|-|/|*

3 Arbre de dérivation

On appelle ***arbre de dérivation*** (ou arbre syntaxique) tout arbre tel que

- la racine est l'axiome
- les feuilles sont des unités lexicales ou ε
- les noeuds sont des symboles non-terminaux

- les fils d'un nœud α sont $\beta_0 \ldots \beta_n$ si et seulement si $\alpha \rightarrow \beta_0 \ldots \beta_n$ est une production (Le parcours préfixe de l'arbre donne le mot)

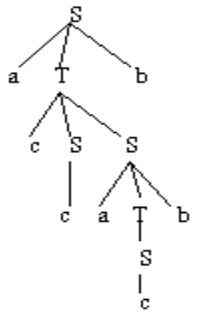
Exemple

Soit la grammaire ayant *S* pour axiome et pour règles de production P:

S→aTb|c

T→cSS|S

Un arbre de dérivation pour le mot ***accacbb*** est :



$S \rightarrow aTb \rightarrow acSSb \rightarrow accSb \rightarrow accaTbb \rightarrow accaSbb \rightarrow accacbb$ (dérivations **gauches**)
ou $S \rightarrow aTb \rightarrow acSSb \rightarrow acSaTbb \rightarrow acSaSbb \rightarrow acSacbb \rightarrow accacbb$ (dérivations **droites**)
Ces deux suites **différentes** de dérivations donnent le **même** arbre de dérivation.

4 Grammaires *ambiguës*

Une grammaire est dite ***ambiguë*** s'il existe un mot de $L(G)$ ayant plusieurs arbres syntaxiques.
Exemple de grammaire ambiguë : Soit *G* donnée par:
instr→ if (expr) instr else instr
inst→ if (expr) instr if (expr) instr
inst→ …
expr→…
Cette grammaire est ambiguë car le mot ***m*= if (*x*>10) if (*y*<0) *a*=1 else *a*=0** possède deux arbres syntaxiques différents :

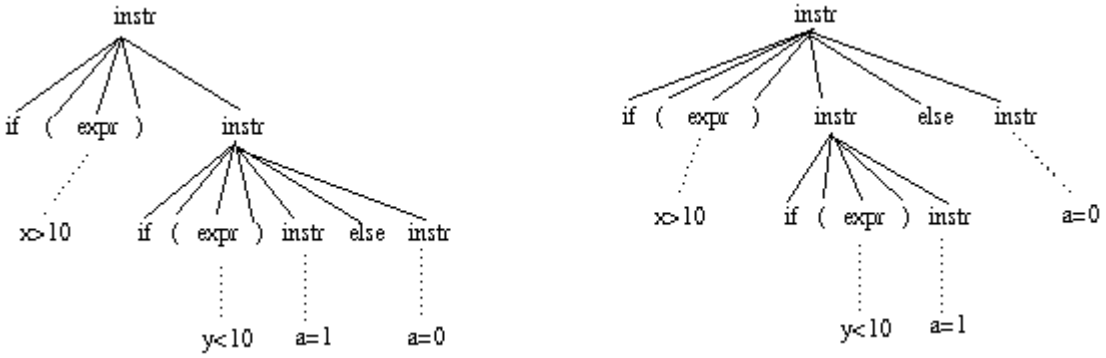


Figure 3.1 : deux arbres différents pour la même chaine.

Car il y a deux interprétations syntaxiques possibles :

```
if (x>10)                  if (x>10)
   if (y<0)                     if (y<0)
      a=1                          a=1
   else                         // finsi
```

```
            a=0                 else
          // finsi                   a=0
        // finsi                 // finsi
```

Solution proposée

stmt → matched-stmt
| open-stmt
matched-stmt→ if expr then matched-stmt else matched-stmt
| other
open-stmt → if expr then stmt
| if expr then matched-stmt else open-stmt

5 Mise en œuvre d'un analyseur syntaxique

Il existe deux approches (deux méthodes) pour construire cet arbre de dérivation : une méthode descendante et une méthode ascendante.

Chapitre 04: Analyse syntaxique descendante

5.1 Analyse descendante

Le terme "LL(1)" signifie que l'on parcourt l'entrée de gauche à droite (L pour Left to right scanning), que l'on utilise les dérivations gauches (L pour Leftmost derivation), et qu'un seul symbole de prévision est nécessaire à chaque étape nécessitant la prise d'une décision d'action d'analyse (1). On appelle grammaire LL(1) une grammaire pour laquelle la table d'analyse décrite précédemment n'a aucune case définie de façon multiple.

Exemple soit la grammaire des expressions arithmétiques classiques Gexp.

1- Expr → Expr + Term
2- | Expr - Term
3- | Term
4-Term →Term * Factor
5- | Term * Factor
6- | Factor
7-Factor → (Expr)
8- | nb
9- | id

Considérons que l'on ait à analyser syntaxiquement (**parser**) l'expression **x-2*y** avec la grammaire des **expressions arithmétiques classiques** (Gexp). On va utiliser le signe → dans la colonne des numéros de règle pour indiquer que l'on avance le pointeur de lecture. La flèche verticale ↑ indiquera la position du pointeur de lecture. Si l'on applique bêtement la méthode expliquée ci-dessus, on va pouvoir tomber sur ce type de dérivation :

Règle	Phrase	Entrée
	E	↑x-2*y
1	*E* + *T*	↑x-2*y
1	*E*+*T*+*T*	↑x-2*y
1	*E*+*T*+..	↑x-2*y
1	…	↑x-2*y

En fait la grammaire telle qu'elle est définie pose un problème intrinsèque car elle est récursive à gauche. Ignorons pour l'instant ce problème et supposons que l'analyseur syntaxique ait effectué la dérivation suivante :

Règle	Phrase	Entrée
	E	↑x-2*y
3	*T*	↑x-2*y
6	*F*	↑x-2*y
9	*id*	↑x-2*y
→	*id*	x↑-2*y

Ici, on ne boucle pas mais on n'a pas reconnu la phrase entière, on en a reconnu une partie, mais on n'a plus de non terminal à remplacer, on ne sait pas si l'on a fait un mauvais choix à un moment ou si la phrase n'est pas valide. Comme il reste un caractère à droite du →, on s'est sans doute planté, donc il faut un mécanisme de gestion des retours arrière (a backtracker) et, en supposant que l'on puisse essayer toutes les possibilités, on arrivera finalement à la dérivation suivante :

Règle	Phrase	Entrée
	E	↑x-2*y
2	*E*-*T*	↑x-2*y
3	*T*-*T*	↑x-2*y
6	*F*-T	↑x-2*y
9	*Id*-*T*	↑x-2*y

→	*Id-T*	x↑-2*y
→	*Id-T*	x-↑2*y
4	*Id-T*F*	x-↑2*y
6	*Id-F*F*	x-↑2*y
8	*Id-nb*F*	x-↑2*y
→	*Id-nb*F*	x-2↑*y
→	*Id-nb*F*	x-2*↑y
9	*Id-nb*id*	x-2*↑y
→	*Id-nb*id*	x-2*y↑

5.2 Grammaire LL(1)

Le terme "LL(1)" signifie que l'on parcourt la chaine d'entrée de gauche à droite (L pour Left to right scanning), que l'on utilise les dérivations gauches (L pour Leftmost derivation), et qu'un seul symbole de prévision est nécessaire à chaque étape nécessitant la prise d'une décision d'action d'analyse (1). On appelle grammaire LL(1) une grammaire pour laquelle la table d'analyse décrite précédemment n'a aucune case définie de façon multiple.
Si une grammaire est LL(1), l'analyse syntaxique peut se faire par l'analyse descendante. Mais

comment savoir que notre grammaire est LL(1) ?

Etant donnée une grammaire :
1. La rendre non ambiguë : Il n'y a pas de méthodes. Une grammaire ambiguë est une grammaire qui a été mal conçue.
2. Eliminer la récursivité à gauche si nécessaire.
3. La factoriser à gauche si nécessaire
4. Construire la table d'analyse

5.2.1 Récursivité à gauche

Une grammaire récursive à gauche peut faire boucler un analyseur réalisé par descente récursive, même si ce dernier possède un mécanisme de gestion des retours arrière comme évoqué plus haut. En effet, en essayant de développer le non-terminal A, on peut éventuellement se retrouver en train de développer A de nouveau, et cela sans avoir consommé de symbole en entrée.
Donc, dans la plupart des cas, en écrivant soigneusement une grammaire, après avoir éliminé les récursivités à gauche et l'avoir factorisée à gauche, nous pouvons obtenir une grammaire qui peut être analysée par descente récursive sans rebroussement.

Définition

Une grammaire **G** est **Récursive Gauche** (**RG**) si il existe une dérivation de la forme : $A \rightarrow^{+} Aw$, avec $A \in \mathbf{N}$ et $w \in (\mathbf{N} \cup \mathbf{T})^*$
Une récursivité **A → Aw** est dite **immédiate**
Exemple
S →ScA|B
A→Aa|ε
B→Bb|d|e

Cette grammaire contient plusieurs récursivités à gauche immédiates.

Elimination de la récursivité à gauche immédiate :

Remplacer toute règle de la forme **A→Aα|β** par les deux règles :
A→βA'
A'→ αA'| ε

Sur l'exemple, on obtient :
S →|BS'
S'→cAS'|ε
A→A'
A'→aA'|ε
B→dB'|eB'
B→bB'|ε
C'est à dire en fait que **S→Sα|ε** est équivalent à **S→αS|ε** (qui, elle, n'est pas immédiatement récursive à gauche, mais à droite).

Exemple

S→**Aa|b**

A→Ac|Sd|c
Le non-terminal *S* est récursif à gauche car S→**Aa** S→**Sda** (mais il n'est pas immédiatement récursif à gauche).

Elimination de la récursivité à gauche :

Ordonner les non-terminaux $A_1,A_2,\ldots A_n$
Pour i=1 à n faire
  pour j=1 à i-1 faire
    remplacer chaque production de la forme $\mathbf{A_i \rightarrow A_j\alpha}$ **où** $\mathbf{A_j \rightarrow \beta_1|\ldots|\beta_p}$ par $\mathbf{A_j \rightarrow \beta_1\alpha|\ldots|\beta_p\alpha}$
  fin pour
  éliminer les récursivités à gauche immédiates des productions A_i
fin pour

Exemple Soit la grammaire suivante:
S→Aa|b
A→Ac|Sd|BA|c
B→SSc|a

On ordonne ***S,A*,B** :
S→Aa|b
i=1 pas de récursivité immédiate dans **S→Aa|b**
i=2 et j=1 on obtient **A→Ac|Aad|bd|BA|c**
  On élimine la récursivité immédiate :
    A→bdA'|BAA'|cA'
    A'→ cA'|adA'|ε
i=3 et j=1 on obtient **B→AaSc|bSc|a**
  et j=2 donne **B→ bdA'aSc| BAA'aSc| cA'aSc|bSc|a**
  on élimine la récursivité immédiate: **B→ bdA'aScB' | cA'aScB' |bScB' |a B'**
    B'→ AA'aScB' | ε

on a obtenu:
S→Aa|b
A→bdA'|BAA'|cA'
A'→ cA'|adA'|ε
B→ bdA'aScB' | cA'aScB'|bScB'|aB'
B'→ AA'aScB'| ε

5.2.2 Factorisation à gauche

L'idée de base est que pour développer un non-terminal *A* quand il n'est pas évident de choisir l'alternative à utiliser (c à d quelle production prendre), on doit réécrire les productions de *A* de façon à **différer** la décision jusqu'à ce que suffisamment de texte ait été lu pour faire le bon choix.

Exemple
S→bacdAbd|bacdBcca
A→ az
B→cz
Au départ, pour savoir s'il faut choisir **S→bacdAbd** ou **S→bacdBcca** , il faut avoir lu la 5ème lettre du mot (un **a** ou un **c**). On ne peut donc pas dès le départ savoir quelle production prendre.

Algorithme de factorisation à gauche

Pour chaque non-terminal *A*
trouver le plus long préfixe α commun à deux de ses alternatives ou plus
Si $\alpha \neq \varepsilon$, remplacer $A \rightarrow \alpha\beta_1|\ldots|\alpha\beta_n|\gamma_1|..|\gamma_p$ (où les γ_i ne commencent pas par α) par le deux règles:
$A \rightarrow \alpha A'|\gamma_1|..|\gamma_p$
$A' \rightarrow \beta_1|\ldots|\beta_n$
finpour
Recommencer jusqu'à ne plus en trouver.

Exemple

S→aEbS|aEbSeB|a
E→bcB|bca
B→ba
Factorisée à gauche, cette grammaire devient :
S→aS''
S''→EbSS'|ε
S'→eB| ε
E →bcE'
E'→B|a
B→ba

5.2.3 Table d'analyse LL(1)

Pour construire une table d'analyse, on a besoin des ensembles PREMIER et SUIVANT

Calcul de PREMIER

Pour toute chaîne $\alpha \in (V_t \cup V_n)^*$, on cherche PREMIER($\alpha$): l'ensemble de tous les **terminaux** qui peuvent **commencer** une chaîne qui se dérive de α, C'est à dire que l'on cherche toutes les lettres *a* telles qu'il existe une dérivation $\alpha \rightarrow \alpha^* \beta$ ($\beta\ \alpha \in (V_t \cup V_n)^*$,).

Algorithme de construction des ensembles PREMIER :

1. Si *X* est un non-terminal et $X \rightarrow Y_1, Y_2 \ldots Y_n$ est une production de la grammaire (avec Y_i symbole terminal ou non-terminal) alors:
   - ajouter les éléments de PREMIER(Y_1) **sauf** ε dans PREMIER(*X*)
   - s'il existe j (j ∈ **{2,…,n}**) tel que pour tout i=1,…,j-1 on a $\varepsilon \in$ PREMIER(Y_i), alors ajouter les éléments de PREMIER(Y_j) **sauf** ε dans PREMIER(X)
   - si pour tout i=1,…, n, $\varepsilon \in$ PREMIER(Yi), alors ajouter ε dans PREMIER(X)
2. Si X est un non-terminal et $X \rightarrow \varepsilon$ est une production, ajouter ε dans PREMIER(X)

3. Si X est un terminal, PREMIER(X)={X}
4. Recommencer jusqu'à ce qu'on n'ajoute rien de nouveau dans les ensembles PREMIER.

Exemple

E→TE'
E'→+TE'|ε
T→FT'
T'→*FT'|ε
F→(E)|nb

PREMIER(E) = PREMIER(T)={(,nb}
PREMIER(E') ={+,ε}
PREMIER(T) = PREMIER(F) ={(,nb}
PREMIER(T') ={*, ε }
PREMIER(F) ={(,nb}

Exemple 2 :
S→ABC
A→aA|ε
B→bB|cB| ε
C→de|da|dA

PREMIER(S)={a,b,c,d}
PREMIER(B) ={b,c, ε }
PREMIER(A) ={a, ε }
PREMIER(T') ={d}

Calcul de SUIVANT

Pour tout non-terminal A, on cherche SUIVANT(*A*) : l'ensemble de tous les symboles terminaux *a* qui peuvent apparaître immédiatement à droite de *A* dans une dérivation : **S→*αAaβ**

Exemple
S→Sc|Ba
B→Pa|bPb|P|ε
p→dS
a,b,c ∈ SUIVANT(*S*) car il y a les dérivations S→*Sc, S→*dSa, et S→*bdSba...

Algorithme de construction des ensembles SUIVANT :

1. Ajouter un marqueur de fin de chaîne (symbole $ par exemple) à SUIVANT(*S*) (*S* est l'axiome)
2. S'il y a une production **A→αBβ** où *B* est un non-terminal, alors ajouter le contenu de PREMIER(β) à SUIVANT(*B*), **sauf ε**
3. S'il y a une production **A→αB**, alors ajouter SUIVANT(*A*) à SUIVANT(*B*)
4. S'il y a une production **A→αBβ** avec ε ∈ PREMIER(β), alors ajouter SUIVANT(*A*) à SUIVANT(*B*).
5. Recommencer à partir de l'étape 3 jusqu'à ce qu'on n'ajoute rien de nouveau dans les ensembles SUIVANT.

Exemple 1 :
E→TE'
E'→+TE'|ε
T→FT'
T'→*FT'|ε
F→(E)|nb

Suivant(E)= Suivant(E')={$,)}
Suivant(T)= Suivant(T')={ +,),$}
Suivant(F)={ *,+,),$}

Exemple 2 :
S → aSb | cd | SAe
A →aAdB |ε
B → bb

	PREMIER	SUIVANT
S	***a c***	***$ b a e***
A	***a ε***	***e d***
B	***b***	***e d***

Construction de la table d'analyse

Une table d'analyse est un tableau M à deux dimensions qui indique pour chaque symbole non-terminal *A* et chaque symbole terminal *a* ou symbole $ la règle de production à appliquer.

- Pour chaque production $A \rightarrow \alpha$ faire
  1. pour tout a ∈ PREMIER(α) (et a ≠ ε), rajouter la production **A→α** dans la case M[A,a]
  2. si ε ∈ PREMIER(α), alors pour chaque b ∈ SUIVANT(*A*) ajouter **A→α** dans *M*[*A*,*b*]
- Chaque case M[A,a] vide est une erreur de syntaxe

Avec l'exemple 1, on obtient la table

	nb	**+**	*****	**(**	**)**	**$**
E	**E→TE'**			**E→TE'**		
E'		**E'→+TE'**			**E'→ε**	**E'→ε**
T	**T→FT'**			**T→FT'**		
T'		**T'→ε**	**T'→*FT'**		**T'→ε**	**T'→ε**
F	**F→nb**			**F→(E)**		

5.2.4 Analyseur syntaxique

Maintenant qu'on a la table, comment l'utiliser pour déterminer si un mot *m* donné est tel que **S→*m?**
On utilise une **pile**.

Algorithme :

Données : mot *m*, table d'analyse *M* et un pointeur ***ps*** sur la 1ère lettre de *m*

Initialization de la pile : $S \$$

Répéter
  Soit *X* le symbole en sommet de pile
  Soit *a* la lettre pointée par *ps*
  Si *X* est un non terminal **alors**
    Si M[A,a]= X→Y1…Yn alors
      enlever *X* de la pile
      mettre Y_n puis Y_{n-1} puis ...puis Y_1 dans la pile
      émettre en sortie la production $X \rightarrow Y_1 \ldots Y_n$
    sinon
      ERREUR
    finsi
  Sinon **Si** *X*=$ **alors**
      Si *a*=$ **alors** ACCEPTER
      Sinon ERREUR
    Sinon **Si** *X*=*a* **alors**
      enlever *X* de la pile
      avancer *ps*
      Sinon ERREUR
    finsi
  finsi
jusqu'à ERREUR ou ACCEPTER

Sur notre exemple (grammaire *E*,*E'*,*T*,*T'*,*F*), soit le mot *m*=3+4*5

PILE	Entrée	Sortie
$ *E*	3+4*5$	**E→TE'**
$ *E'T*	3+4*5$	**T→FT'**
$ *E'TF*	3+4*5$	**F→nb**
$ *E'T*3	3+4*5$	
$ *E'T'*	+4*5$	**T'→ε**
$ *E'*	+4*5$	**E'→+TE'**
$ *E'T*+	+4*5$	

$ *E'T*	4*5$	**T→FT'**
$ *E'TF*	4*5$	**F→nb**
$ *E'T*4	4*5$	
$ *E'T'*	*5$	**T'→*FT'**
$ *E'TF**	*5$	
$ *E'TF*	5$	**F→nb**
$ *E'T*5	5$	
$ *E'T'*	$	**T'→ε**
$ *E'*	$	**E'→ε**
$	$	**analyse syntaxique réussie**

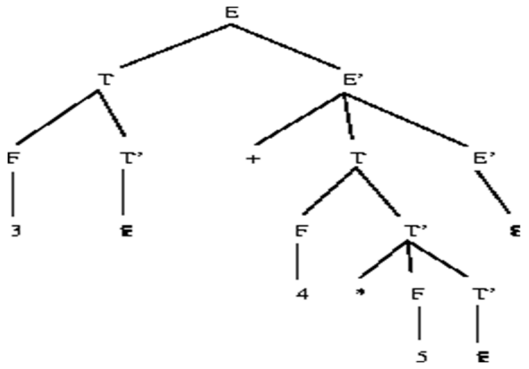


Figure 4.1: Arbre syntaxique résultants de l'analyse de l'expression 3+4*5$

Si l'on essaye d'analyser maintenant le mot ***m*=(7+3)5**

PILE	Entrée	Sortie
$ *E*	(7+3)5$	**E→TE'**
$ *E'T*	(7+3)5$	**T→FT'**
$ *E'TF*	(7+3)5$	**F→(nb)**
$ *E'T*) *E* (	(7+3)5$	
$ *E'T*) *E*	7+3)5$	**E→TE'**
$ *E'T*)*E'T*	7+3)5$	**T→FT'**
$ *E'T*)*E'TF*	7+3)5$	**F→7**
$ *E'T*)*E'T*7	7+3)5$	
$ *E'T*)*E'T'*	+3)5$	**T'→ε**
$ *E'T*)*E'*	+3)5$	**E'→+TE'**
$ *E'T*)*E'T*+	+3)5$	
$ *E'T*)*E'T*	3)5$	**T→FT'**
$ *E'T*)*E'TF*	3)5$	**F→3**
$ *E'T*)*E'T*3	3)5$	
$ *E'T*)*E'T'*	)5$	**T'→ε**
$ *E'T*)*E'*	)5$	**E'→ ε**
$ *E'T*)	)5$	
$ *E'T*	5$	**ERREUR !!**

Donc ce mot n'appartient pas au langage généré par cette grammaire.

6 Exercices

Exercice 1 :

Supprimer la récursivité gauche des grammaires suivantes:

1. **S⟶A a |b**
   A⟶A c| S d| c

2. **S⟶S a| T S c |d**
   T⟶TbT|ε

Exercice 2 :

On considère la grammaire G définie par :

S⟶ S∨ S| S∧ S|¬ S| (S) | b

1. Montrer que cette grammaire est ambigüe
2. Construire une grammaire équivalente non ambigüe, avec les règles de priorité suivantes : l'opérateur de négation ¬ est le plus prioritaire. L'opérateur ∧ a la seconde plus haute priorité et est associatif à gauche L'opérateur ∨ est le moins prioritaire et est associatif à gauche.

Exercice 3 :

Soit G = ({X,A,B,C,D}, {a,b, c, d},X) avec :

X→AB
A→CD
B→c |ε
C→aCd |ε
D→bbD |ε

1. Calculer les ensembles **Premier** et **Suivant** de la grammaire.
2. Construisez la table d'analyse prédictive de G. Cette grammaire est-elle LL(1) ?
3. Analysez le mot adbb$.

Exercice 4:

Soit la grammaire G :

S → iBae
B → TB | ε
T → [eD] | di
D → ed | ε

1. Calculer les ensembles Premier et Suivant de la grammaire.
2. Calculer la table d'analyse LL(1) pour G.

Exercice 5 :

Soit G = ({S,T} {"a","b", ", "; "(", ")"}, S}) la grammaire suivante

S → a | b|(T)
T→T,S | S

1. G est-elle LL(1) ?
2. Eliminer la récursivité à gauche et factoriser si nécessaire.
3. La nouvelle grammaire est-elle LL(1) ?

Chapitre 05: Analyse syntaxique ascendante

1 Objectif

L'objectif de ce type d'analyse est de Construire un arbre de dérivation du bas (les feuilles, ou les unités lexicales) vers le haut (la racine, ou l'axiome de départ).
Le modèle général utilisé est le modèle par **décalages–réductions**. C'est à dire que l'on ne s'autorise que deux opérations :
Décalage (shift) : décaler d'une lettre le pointeur sur le mot en entrée
Réduction (reduce) : réduire une chaîne (suite consécutive de terminaux et non terminaux à gauche du pointeur sur le mot en entrée et finissant sur ce pointeur) par un non-terminal en utilisant une des règles de production

Exemple Soit la grammaire

S→aSbS|c
avec le mot *u=**aaacbaacbcbcbcbacbc***
*u=**a**aacbaacbcbcbcbacbc* on ne peut rien **réduire**, donc on **décale**
*u=a**a**acbaacbcbcbcbacbc* on ne peut rien réduire, donc on décale
*u=aa**a**cbaacbcbcbcbacbc* on ne peut rien réduire, donc on décale
*u=aaa**c**baacbcbcbcbacbc* ah ! On peut réduire par **S→c**

*u=aaa**S**baacbcbcbcbacbc* on ne peut rien réduire, donc on décale
...
*aaaSbaa**S**bcbcbcbacb*c on peut utiliser **S→c**
*aaaSbaa**S**bcbcbcbacb*c on ne peut rien réduire, donc on décale
*aaaSbaaS**b**cbcbcbacb*c on ne peut rien réduire, donc on décale
*aaaSbaaSb**c**bcbcbacb*c On peut utiliser **S→c**
*aaaSbaaSb**S**bcbcbacbc* On peut utiliser **S→aSbS**
*aaaSba**S**bcbcbacb*c décalage
*aaaSbaS**b**cbcbacb*c décalage
*aaaSbaSb**c**bcbacb*c réduction par **S→c**
*aaaSba**S**b**S**bcbacb*c réduction par **S→aSbS**
*aaaSb**S**bcbacb*c réduction par **S→aSbS**
*aa**S**bcbacb*c décalage
*aaS**b**cbacb*c décalage
*aaSb**c**bacb*c réduction par **S→c**
*aaSb**S**bacb*c réduction par **S→aSbS**
*a**S**bacb*c décalage
*aS**b**acb*c décalage
*aSb**a**cb*c décalage
*aSba**c**b*c réduction par **S→c**
*aSba**S**b*c décalage
*aSba**S**b*c décalage
*aSba**S**b***c** réduction par **S→c**
*a**S**ba**S**b***S** réduction par **S→aSbS**
*a**S**b**S*** réduction par
terminé.

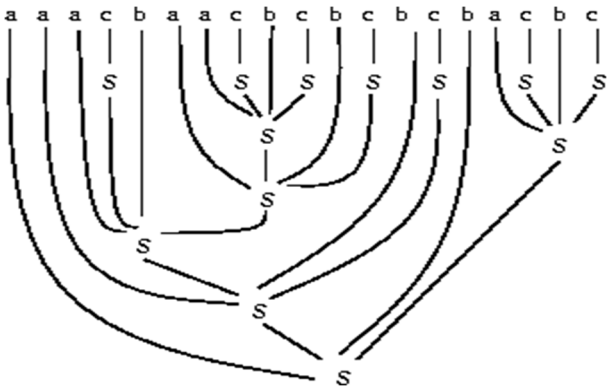


Figure 5.1 : Arbre résultant de l'analyse ascendante

➔ Ça serait bien d'avoir une table qui nous dit si on décale ou si on réduit, et par quoi, lorsque le pointeur est sur une lettre donnée.

2 Analyse ascendante SLR(1)

- C'est une sorte d'automate, qu'on appelle *automate à pile*. Cette table va nous dire ce qu'il faut faire quand on lit une lettre ***a*** et qu'on est dans un état ***i***.
- Soit on **décale**. Dans ce cas, on empile la lettre lue et on va dans un autre état *j*. Ce qui sera noté **d*j***
- Soit on **réduit** par la règle de production numéro ***p***, c à d qu'on remplace la chaîne en sommet de pile par le non-terminal de la partie gauche de la règle de production, et on va dans l'état *j* qui dépend du non-terminal en question. On note ça **r*p***.
- Soit on **accepte** le mot. Ce qui sera noté ACC.
- Soit c'est une **erreur**. c à d Case vide.

Utilise aussi les ensembles **SUIVANT** et **PREMIER**, plus ce qu'on appelle des fermetures de **0-items.** Un 0-item (ou *item*) est une production de la grammaire avec un "**.**" quelque part dans la partie droite. Par exemple E → E . + T ou encore T→ F. ou encore F → . (E)

Fermeture d'un ensemble d'items *I* :

1- Mettre chaque item de *I* dans Fermeture(*I*)
2- Pour chaque item *i* de Fermeture(*I*) de la forme **A→α. *B* β**
  Pour chaque production ***B*** → γ_i
    rajouter l'item ***B*** →.γ_i dans Fermeture (*I*)
3- Recommencer 2 jusqu'à ce qu'on n'ajoute rien de nouveau

Exemple

Soit la grammaire des expressions arithmétiques

E → E+ T
E → T
T →T* F
T → F
F → (E)
F → nb

Et soit l'ensemble d'items **{ T →T * . F, E→E.+T}.** La fermeture de cet ensemble d'items est :
{T→T*.F, E→E.+T, F→.nb, F→.(E)}

Transition par X d'un ensemble d'items I :

$\Delta(I,X)$=Fermeture (tous les items A→α*X* . β où A→α. *X*β est dans *I*)

Sur l'exemple ETF : soit I={T→T*.F, E→E.+T, F→.nb, F→.(E)} on aura
$\Delta(I,F) = \{ T \rightarrow T*F. \}$
$\Delta(I,+) = \{ E \rightarrow E+.T,\ T \rightarrow .T*F,\ T \rightarrow .F,\ F \rightarrow .nb, F \rightarrow .(E) \}$

Collection des items d'une grammaire :

0- Rajouter l'axiome S' avec la production $S' \rightarrow S$
1- Mettre dans l'item I_0 la Fermeture de $S' \rightarrow .S\})$
2- Mettre I_0 dans Collection
3 - Pour chaque I dans Collection faire
  Pour chaque X tel que $\Delta(I, X)$ est non vide
    ajouter ce $\Delta(I, X)$ dans Collection
  Fin pour
4 - Recommencer 3 jusqu'à ce qu'on n'ajoute rien de nouveau

Exemple

I_0={S→.E, E →.E+T, E →.T, T→.T*F, T→. F, F →.nb, F →.(E) }
$\Delta(I_0,E)$= {S→E., E →E.+T}=I_1
$\Delta(I_0,T)$= {E → T., T→T.*F}=I_2
$\Delta(I_0,F)$= {T→ F.}=I_3
$\Delta(I_0,()$= {F→(.E), E→.E+T, E→.T, T→.T*F, T→.F, F→.nb, F→.(E)}=I_4
$\Delta(I_0,nb)$={ F →nb.}=I_5
$\Delta(I_1,+)$= {E →E+.T, T→.T*F, T→. F, F →.nb, F →.(E) }=I_6
$\Delta(I_2,*)$={ T→T*.F, F →.nb, F →.(E)}=I_7
$\Delta(I_4, E)$= {F→(E.), E→E.+T}=I_8
$\Delta(I_4, T)$={ E→T., T→T.*F }=I_2
$\Delta(I_4, F)$={ T→F. }=I_3
$\Delta(I_4,nb)$={ F→nb. }=I_5
$\Delta(I_4,()$={ F→(.E), E→.E+T, E→.T, T→.T*F, T→.F, F→.nb, F→.(E)}=I_4
$\Delta(I_6, T)$= {E →E+T., T→T.*F }=I_9
$\Delta(I_6, F)$={ T→F. }=I_3
$\Delta(I_6, nb)$={ F→nb. }=I_5
$\Delta(I_6, ()$={ F→(.E), E→.E+T, E→.T, T→.T*F, T→.F, F→.nb, F→.(E)}=I_4
$\Delta(I_7, F)$={ T→T*F.}=I_{10}
$\Delta(I_7, nb)$={ F→nb. }=I_5
$\Delta(I_7, ()$={ F→(.E), E→.E+T, E→.T, T→.T*F, T→.F, F→.nb, F→.(E)}=I_4
$\Delta(I_8,))$= {F→(E).}=I_{11}
$\Delta(I_8, +)$= {F→E+.T, T→.T*F, T→. F, F →.nb, F →.(E) }=I_6
$\Delta(I_9,*)$={ T→T*.F, F →.nb, F →.(E)}=I_7

Construction de la table d'analyse SLR :

1- Construire la collection d'items $\{I_0, \ldots I_n\}$
2- l'état i est construit à partir de I_i :
  a) pour chaque $\Delta(I_i, a) = I_j$: mettre **décalage par** j dans la case M[i,a]
  b) pour chaque $\Delta(I_i, A) = I_j$: mettre **aller à** j dans la case M[i,A]
  c) pour chaque $A \rightarrow \alpha.$ contenu dans I_i :
    pour chaque a de SUIVANT(A) faire
      mettre **réduction par** numéro (de la règle $A \rightarrow \alpha$) dans la case M[i,a]

Toujours le même Exemple il nous faut les SUIVANT et PREMIER :

	PREMIER	SUIVANT
E	**nb (**	**$ +)**
T	**nb (**	**$ + *)**
F	**nb (**	**$ + *)**

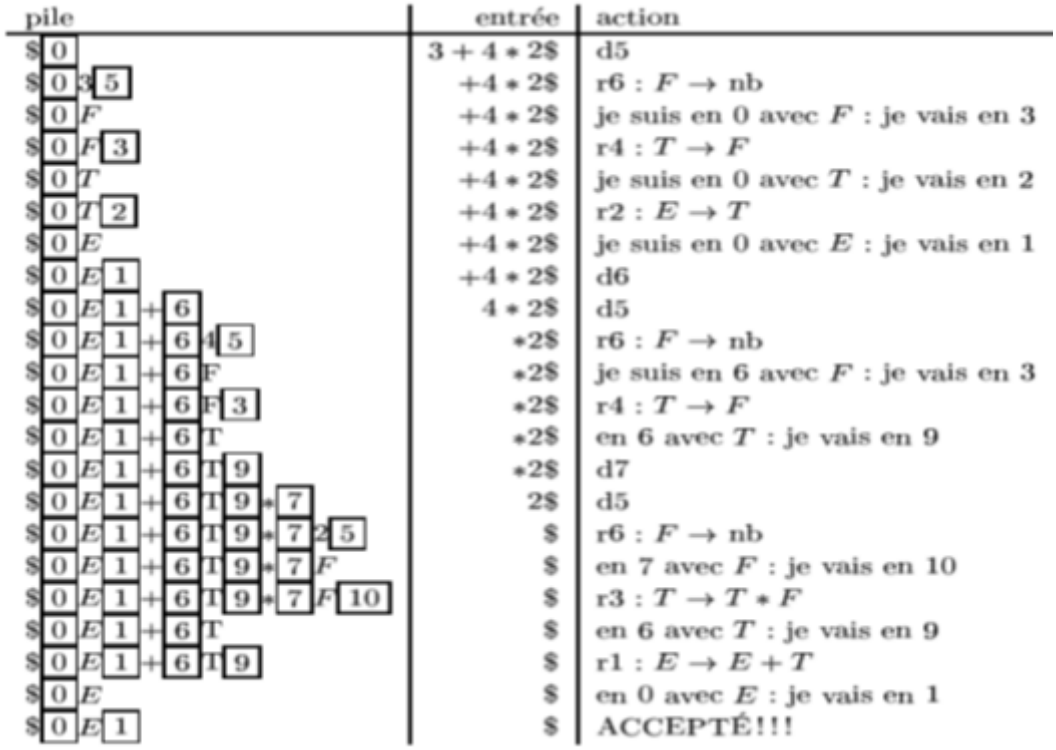
Avec notre exemple ETF, on obtient la table d'analyse LR

état	nb	+	*	(	)	$	E	T	F
0	d5			d4			1	2	3
1		d6				ACC			
2		r2	d7		r2	r2			
3		r4	r4		r4	r4			
4	d5			d4			8	2	3
5		r6	r6		r6	r6			
6	d5			d4				9	3
7	d5			d4					10
8		d6			d11				
9		r1	d7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Analyseur syntaxique SLR

On part dans l'état 0, et on empile et dépile non seulement les symboles (comme lors de l'analyseur LL) mais aussi les états successifs.

Exemple
Le processus de l''analyse du mot m=3+*4$ sera comme suit :



Remarques

1. Cette méthode permet d'analyser plus de grammaires que la méthode descendante (car il y a plus de grammaires SLR que LL) parce que cette méthode d'analyse n'a strictement aucune importance que la grammaire soit récursive à gauche, même au contraire, on **préfère**.
2. Les grammaires ambiguës provoquent des **conflits**
   a. Conflit **décalage/réduction** : on ne peut pas décider à la lecture du terminal *a* s'il faut réduire une production **S→α** ou décaler le terminal
   b. Conflit **réduction/réduction** : on ne peut pas décider à la lecture du terminal *a* s'il faut réduire une production **S→α** ou une production T→**β**

   ⇨ On doit alors résoudre les conflits en donnant des **priorités** aux actions (décaler ou réduire) et aux productions.

3 Analyse syntaxique LR(1)

L'analyse LR incorpore les informations supplémentaires requises dans l'état en redéfinissant les configurations pour inclure un symbole terminal en tant que composant ajouté. Les configurations LR(1) ont la forme générale :

$$A \rightarrow X_1...X_i \bullet X_{i+1}...X_j\ ,\ a$$

Cela signifie que nous avons des états correspondant à $X_1...X_i$ sur la pile et nous cherchons à mettre des états correspondant à $X_{i+1}...X_j$ sur la pile puis à réduire, mais seulement si le jeton suivant X_j est le terminal a qui s'appelle le lookahead (caractère de prévision). Le lookahead n'entre en jeu qu'avec les configurations LR(1) avec un point à l'extrémité droite :

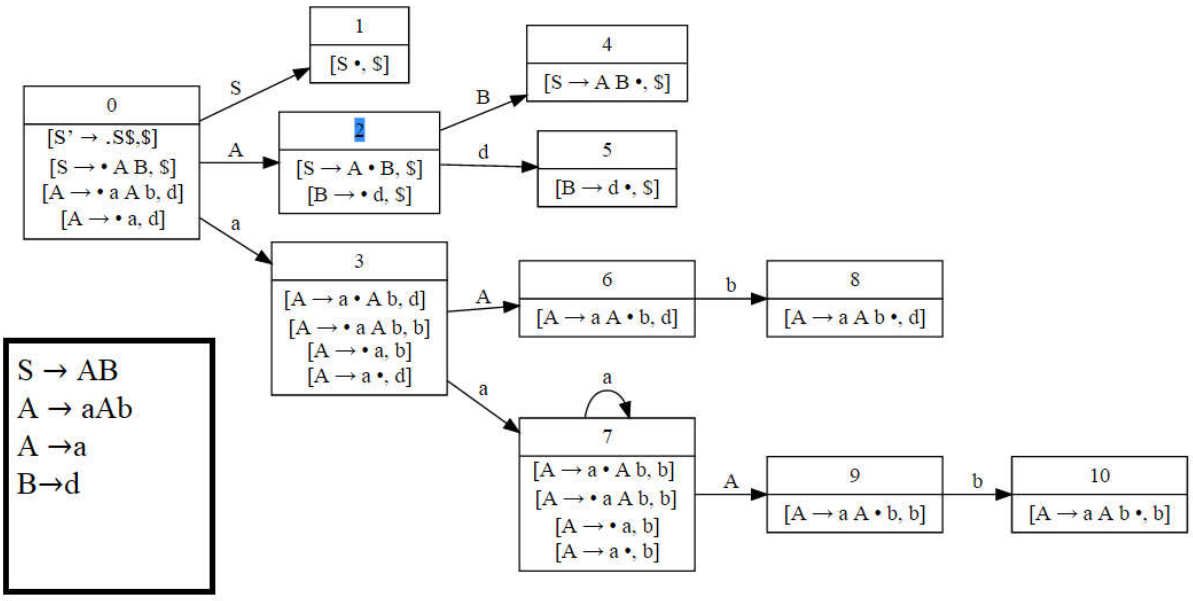
$$A \rightarrow X_1.....X_j\bullet\ ,\ a$$

Cela signifie que nous avons des états correspondant à $X_1.....X_j$ dans le sommet de la pile mais nous ne pouvons réduire que lorsque le symbole suivant est *a*. Le symbole *a* peut-être un terminal, ou $ (marqueur de fin de la chaine). Avec l'analyse SLR(1), nous réduirions si le caractère suivant était l'un de ceux de Suivant(A). Avec l'analyse LR(1), nous ne réduisons que si le caractère suivant est exactement a. Nous pouvons avoir plus d'un symbole dans le lookhead. Alors, la configuration :

$$A \rightarrow u\bullet,\ \{a,b,c\}$$

indique qu'il est valide de réduire u à A uniquement si le jeton suivant est égal à a, b ou c. Le Lookhead sera toujours un sous-ensemble de Suivant(A).

Exemple

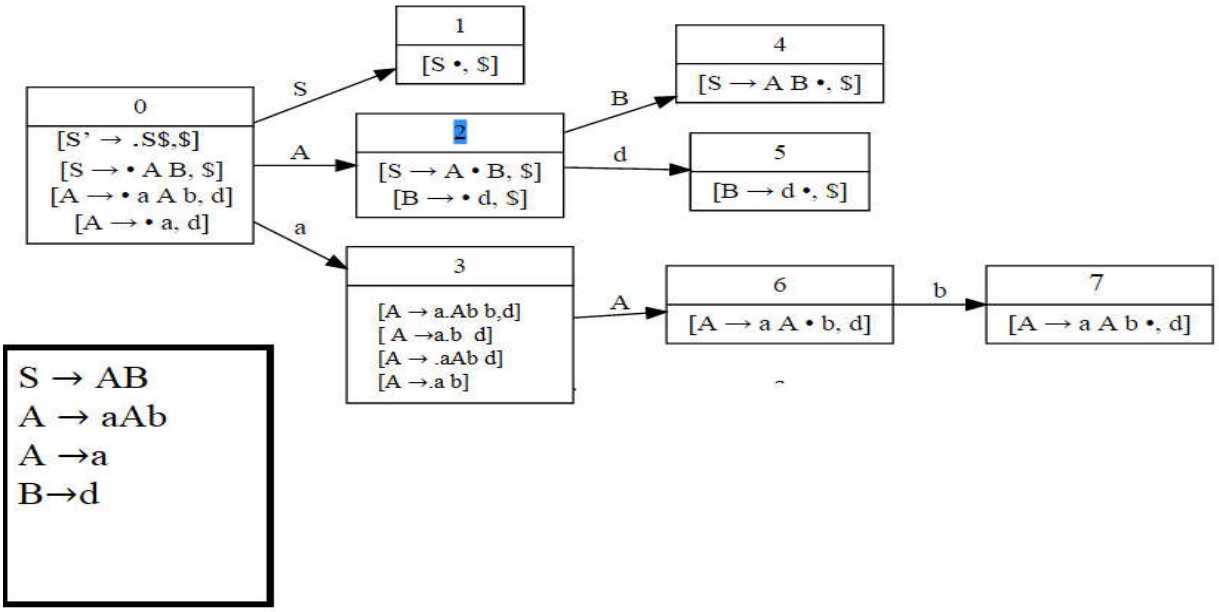


	a	b	d	$	S	A	B
0	**d3**				**1**	**2**	
1				**Acc**			
2			**d5**				**4**
3						**6**	
4				**r1**			
5				**r4**			
6		**d8**					
7	**d7**					**9**	
8			**r2**				
9		**10**					
10		**r2**					

4 Analyse syntaxique LALR(1)

La méthode LALR(1) (LookAhead LR) se base sur la construction de la collection d'ensembles d'items LR(1) ainsi que la table d'analyse LR(1). L'idée générale de cet algorithme est de construire les ensembles d'items LR(1) et de fusionner les ensembles ayant un le caractère de prévision commun. La table d'analyse LALR(1) est construite à partir de la collection des ensembles d'items fusionnés..

Exemple:



	a	b	d	$	S	A	B
0	**d3**				**1**	**2**	
1				**Acc**			
2			**d5**				**4**
3+7						**6**	
4				**r1**			
5				**r4**			
6+9		**d8**					
7	**d7**					**9**	
8			**r2**				
9		**10**					
10		**r2**					

5 L'outil Yacc

Yacc signifie *Yet Another Compiler Compiler*, c'est à dire encore un compilateur de compilateur. Yacc est un outil logiciel qui permet de générer des analyseurs syntaxiques. Pour réaliser un tel analyseur il faut décrire d'abord l'outil à construire dans un fichier **monanalyseur.y**. Ce fichier permet d'obtenir un analyseur syntaxique en langage C qui a pour nom **y.tab.c**. Il convient d'ajouter un fichier lex qui va engendrer l'analyseur lexicale **lex.yy.c** qu'il faudra inclure dans le programme principal. Une fois obtenu cet analyseur il faut le compiler et l'on a alors un exécutable qui effectue l'analyse syntaxique d'un texte suivant les instructions données dans **monanalyseur.y**. Ce fonctionnement peut être schématisé par la figure suivante :

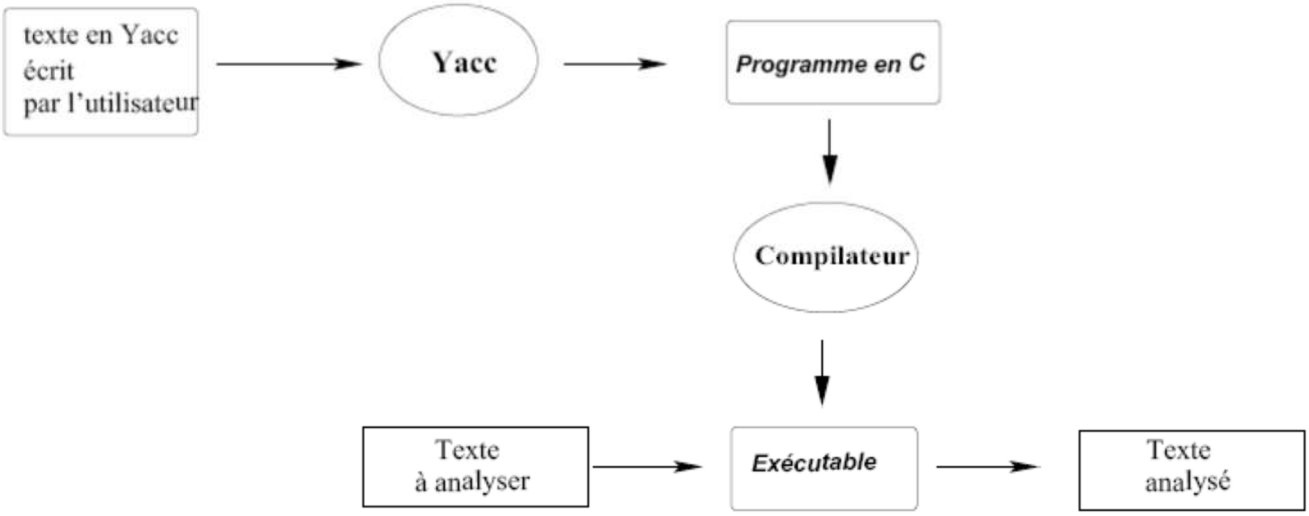


Figure 5.2: Déroulement de l'analyse syntaxique rn utilisant YACC

Les instructions nécessaires à la réalisation sont alors les suivantes
lex monanalyseur.lex
yacc monanalyseur.y
gcc -o monanalyseur y.tab.c

5.1 La structure d'une spécification Yacc

Un programme en Yacc se divise en trois parties séparées par des **%%**. La première contient des options, et des déclarations et initialisations de variables utiles pour le programme en C généré. La seconde contient la grammaire du langage et les actions à effectuer lorsque au cours de l'analyse on a réduit par une règle, enfin la troisième partie contient l'enveloppe du programme C généré.

%{
déclaration (en C) de variables, constantes, inclusions de fichiers,
%}
déclarations des unités lexicales utilisées
déclarations de priorités et de types
%%
règles de production et actions sémantiques
%%
routines C et bloc principal

5.1.1 Partie Déclaration

La partie déclarations contient les déclarations "C" (entre %{ et %}) ainsi que la déclaration des noms de tokens :
%token name1 name2 ...
Les noms qui n'ont pas été déclarés en tant que "nom de token" sont considérés comme des non-terminaux. Lorsque les tokens correspondent à des opérateurs pour lesquels on souhaite spécifier une propriété d'associativité, on utilise *left* et *right* à la place de token.

Exemple on écrira :

```
%left '+' '-'   /* addition et soustraction associatives a gauche */
%right '^'      /* exponentiation associative a droite */
```

5.1.2 Grammaire et actions

Les règles de production sont des suites d'instructions de la forme

```
non-terminal :  prod1
              | prod2
                 ...
              | prodn
             ;
```

Les actions sémantiques sont des instructions en C insérées dans les règles de production. Elles sont exécutées chaque fois qu'il y a réduction par la production associée.
Exemple

```
G : S B 'X' {printf("mot reconnu");}
;
S : A {printf("réduction par A");} T {printf("réduction par T");} 'a'
;
```

5.1.3 Le programme en C

La troisième partie doit contenir le programme principal main() qui doit en général faire un appel à la fonction **yyparse()** qui est créée par Yacc, et il doit aussi contenir un : **#include ``lex.yy.c''.**

5.2 Communication avec l'analyseur lexical : yylval

L'analyseur syntaxique et l'analyseur lexical peuvent communiquer entre eux par l'intermédiaire de la variable *int yylval.*
Dans une action lexicale (donc dans le fichier lex par exemple), l'instruction *return(ul)* permet de renvoyer à l'analyseur syntaxique l'unité lexicale *ul*. La valeur de cette unité lexicale peut être rangée dans *yylval*.
L'analyseur syntaxique prendra automatiquement le contenu de *yylval* comme valeur de l'attribut associé à cette unité lexicale.

La variable yylval est de type YYSTYPE (déclaré dans la bibliothèque yacc/bison) qui est un *int* par défaut. On peut changer ce type par:

```
#define YYSTYPE autre_type_C
```

Ou encore par :
%union {champs d'une union C}
qui déclarera automatiquement YYSTYPE comme étant une telle union.
Par exemple

```
%union {
        int entier;
        double reel;
        char * chaine;
      }
```

permet de stocker dans yylval à la fois des int, des double et des char *.
L'analyseur lexical pourra par exemple contenir

```
{nombre}  {
        yylval.entier=atoi(yytext);
        return NOMBRE;
        }
```

Le type des lexèmes doit alors être précisé en utilisant les noms des champs de l'union

```
%token <entier> NOMBRE
%token <chaine> IDENT CHAINE COMMENT
```

On peut également typer des non-terminaux (pour pouvoir associer une valeur de type autre que int à un non-terminal) par

```
%type <entier> S
%type <chaine> expr
```

5.3 Variables, fonctions et actions prédéfinies

YYPARSE() : appel de l'analyseur syntaxique.
YYACCEPT : instruction permettant de stopper l'analyseur syntaxique.
YYABORT : instruction qui permet également de stopper l'analyseur. yyparse retourne alors 1, ce qui peut être utilisé pour signifier l'echec de l'analyseur.
main() : le main par défaut se contente d'appeler yyparse(). L'utilisateur peut écrire sa propre main dans la partie du bloc principal.
%start non-terminal : action pour signifier quel non-terminal est l'axiome. Par défaut, c'est le premier décrit dans les règles de production.

Exemple

Un exemple simple est le suivant, c'est un analyseur qui lit une suite de a et de b représentant un mot de parenthèse (a ouvrante, b fermante) et qui donne les règles qui l'ont engendré suivant la grammaire:
S →aSbS, S → aSb, S →abS, S→ab
Le point virgule sert de marqueur de fin

```
%token PVIRG PARO PARF
%%
s0 :exp PVIRG {printf("fini\n"); exit(0);}
      ;
exp:    PARO exp PARF exp {printf(" regle1 \n");}
|PARO exp PARF {printf(" regle2 \n");}
|PARO PARF exp {printf(" regle3 \n");}
|PARO PARF {printf(" regle4 \n");}
;
%%
#include <ctype.h>
#include <stdio.h>
#include "lex.yy.c"
main(){
yyparse();
}
yyerror(char *s){
printf ("%s\n",s);}
```

5.4 Etude de cas:

1. Le source Lex du mini-interprète d'expressions

```
%{#include "global.h"
#include "calc.h"
#include <stdlib.h>
%}
blancs    [ \t]+
chiffre   [0-9]
entier    {chiffre}+
exposant  [eE][+-]?{entier}
reel    {entier}("."{entier})?{exposant}?
%%
{blancs}  { /* On ignore */ }
{reel}    {
     yylval=atof(yytext);
     return(NOMBRE);
   }
"+"   return(PLUS);
"-"   return(MOINS);
"*"   return(FOIS);
"/"   return(DIVISE);
"^"   return(PUISSANCE);
"("   return(PARENTHESE_GAUCHE);
")"   return(PARENTHESE_DROITE);
```

```
"\n"  return(FIN);
```

2. Le source Yacc du mini-interprète d'expressions

C'est le plus important, et le plus intéressant. Le voici :

```
%{
#include "global.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
%}
%token  NOMBRE
%token  PLUS  MOINS FOIS  DIVISE  PUISSANCE
%token  PARENTHESE_GAUCHE PARENTHESE_DROITE
%token  FIN
%left PLUS  MOINS
%left FOIS  DIVISE
%left NEG
%right  PUISSANCE
%start Input
%%
Input:
    /* Vide */
  | Input Ligne
  ;
Ligne:
    FIN
  | Expression FIN    { printf("Resultat : %f\n",$1); }
  ;
Expression:
    NOMBRE      { $$=$1; }
  | Expression PLUS Expression {$$=$1+$3; }
  | Expression MOINS Expression {$$=$1-$3; }
  | Expression FOIS Expression {$$=$1*$3; }
  | Expression DIVISE Expression {$$=$1/$3; }
  | MOINS Expression %prec NEG {$$=-$2; }
  | Expression PUISSANCE Expression {$$=pow($1,$3); }
  | PARENTHESE_GAUCHE Expression PARENTHESE_DROITE {$$=$2;}
    ;
%%
int yyerror(char *s) {
  printf("%s\n",s);
}
int main(void) {
  yyparse();
}
```

Remarque

Veuillez noter que le fichier **global.h** contiendra :

```
#define YYSTYPE double
extern YYSTYPE yylval;
```

6 Exercices

Exercice 01: Soit la grammaire G({S,L},{a,b,c},S) :

S→ aLb |ab | c
L→ S | L S

1. Construire la table d'analyse SLR (1) de G. G est-elle SLR(1) ? justifier votre réponse.
2. Construire la table d'analyse LR (0) de G. G est-elle LR(0) ? justifier votre réponse.
3. Analyser la chaine « accaaccabb »

Exercice 02: Soit G la grammaire suivante pour des blocs Java (simplifiés) :

bloc → { inst_l }
inst_l → inst_l inst | ε
inst → decl | exp | bloc

decl et exp sont ici des unités lexicales terminales pour les déclarations et les expressions.

1. Construire l'automate LR pour G.
2. G est-elle LR(0) ? SLR(1) ? Justifier rigoureusement en énumérant tous les conflits et en précisant leur type (shift/reduce ou reduce/reduce).

Exercice 03 : Soit G la grammaire suivante l'analyse de la boucle while do en PASCAL :

E → while E do E
E → id := E
E → E + E
E → id

1. Donner l'automate des items LR(1) canoniques pour G.
2. Donner la table des actions et successeurs LR (1) de G.
3. La grammaire G est-elle LR(1)?
4. Si cette grammaire est de type LALR(1)? Donner le résultat de l'analyse de la chaîne
" w = while id1+id3 do id5:=id4+id3".

Exercice 04 : Soit G la grammaire Yacc suivante :

```
%start E

F : '(' E ')' | 'id' ;
RF : '*' F RF | ;
T : F RF ;
RT : '+' T RT | ;
E : 'id' ':=' E | T RT ;
```

Combien G a-t-elle de conflits LALR(1) ? Les lister.

Chapitre 06 : Analyse sémantique

1 Introduction

Certaines propriétés fondamentales du langage source à traduire ne peuvent être décrites à l'aide de la seule grammaire hors contexte du langage, car on a souvent besoin de plus d'informations qui dépendent du contexte. Par exemple (exemples seulement, car cela dépend du langage source) :
- on ne peut pas utiliser dans une instruction une variable qui n'a pas été déclarée au préalable
- on ne peut pas déclarer deux fois une même variable
- lors d'un appel de fonction, il doit y avoir autant de paramètres formels que de paramètres effectifs, et leur type doit correspondre
- on ne peut pas multiplier un réel avec une chaîne.

D'autre exemple, lorsqu'on rencontre un identificateur, est-ce une fonction ou une variable ? Si c'est une variable, est-elle déclarée ? Est-elle utilisable à cet endroit (dans ce contexte) ? Quel type de valeur est stocké dans cette variable ? …. Si c'est une fonction, combien et quel type d'argument a-t-elle ?
Toutes ces informations sont disponibles dans le programme mais pas forcément à l'endroit où l'on rencontre l'identificateur : on dit qu'elles sont présentes dans le ***contexte***, et on les appellera informations *contextuelles*.
En plus de reconnaître si le programme est bien une phrase du langage, le parseur doit donc construire ce contexte, qui lui permettra ensuite d'avoir accès à ces informations en temps voulu. Le contexte est défini par des déclarations (en général en début de programme/fonction/procédure...) qui donnent le contexte pour les feuilles de l'arbre syntaxique, et par des règles sémantiques qui permettent de déduire le contexte pour tous les noeuds internes de l'arbre.

2 Définition dirigée par la syntaxe

Une définition dirigée par la syntaxe (DDS) est un formalisme permettant d'associer des actions à une production d'une règle de grammaire.
- Chaque symbole de la grammaire (terminal ou non) possède un ensemble d'**attributs (ou des variables)**.
- Chaque règle de production de la grammaire possède un ensemble de **règles sémantiques** qui permettent de calculer la valeur des attributs associés aux symboles apparaissant dans la production.
- Une règle sémantique est une suite d'instructions algorithmiques : elle peut contenir des affectations, des si-sinon, des instruction d'affichage,…

On notera X.a l'attribut *a* du symbole X, s'il y a plusieurs symboles X dans une production, on notera $X^{(0)}$ s'il est en partie gauche, $X^{(1)}$ si c'est le plus à gauche de la partie à droite, $X^{(2)}$ si c'est le deuxième plus à gauche de la partie à droite,…, $X^{(n)}$ si c'est le deuxième plus à droite de la partie à droite.

Exemple soit la grammaire

S→aSb|as|cSacS|ε

Attributs :
- Nba calcul du nombre de a,
- Nbc calcul du nombre de c,

Une DDS permettant de calculer ces attributs pourrait être :

Production	Règle sémantique
S→aSb	S(0).Nba:=S(1).Nba+1 S(0).Nbc:=S(1).Nbc
S→aS	S(0).Nba:=S(1).Nba+1 S(0).Nbc:=S(1).Nbc
S→cSacS	S(0).Nba:=S(1).Nba+ S(1).Nba +1 S(0).Nbc:= S(1).Nbc+ S(1).Nbc+2
S→ε	S(0).Nba:=0 S(0).Nbc:=0
S'→S	//Le résultat final est dans S.Nba et S.Nbc

On peut dessiner un arbre syntaxique avec la valeur des deux attributs pour chaque symbole non terminal,, pour la chaîne: acaacabb

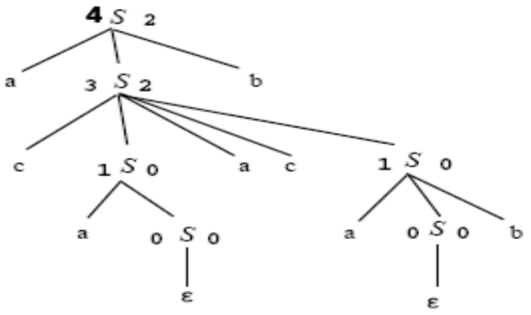


Figure 6.1 : arbre étiqueté avec deux attributs

2.1 Les types des attributs

Une **grammaire attribuée** est donc une grammaire hors contexte dans laquelle on associe à chaque symbole de la grammaire des informations sémantiques. Ainsi, chaque symbole possède un ensemble d'attributs, partitionné en deux sous-ensembles que l'on appelle les **attributs synthétisés** et **les attributs hérités** de ce symbole. Si l'on considère un noeud d'un arbre syntaxique comme une structure avec des champs pour stocker de l'information, alors un attribut correspond à un nom de champ.

2.1.1 Les attributs synthétisés

Un attribut est dit synthétisé lorsqu'il est calculé pour le non terminal de la partie gauche en fonction des attributs des non terminaux de la partie droite. Sur l'arbre, la valeur d'un attribut en un nœud se calcul en fonction des attributs de ses fils, c-à-d le calcul de l'attribut se fait des feuilles vers la racine.

Remarque

Une grammaire attribuée n'ayant que des attributs synthétisés est dite S-attribuée.

Exemple

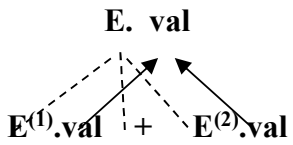


Figure 6.2 : Exemple des attributs synthétisés

2.1.2 Les attributs hérités

Un attribut est dit hérité lorsqu'il est calculé à partir des attributs de la partie gauche, et éventuellement des attributs d'autres non terminaux de la partie droite. Sur l'arbre, la valeur d'un attribut à un nœud

se calcul en fonctions des attributs se fait des frères et du père. C-à-d que le calcul de l'attribut se fait de la racine vers les feuilles.

Remarque

Une grammaire attribuée n'ayant que des attributs hérités et telle que ces attributs ne dépendent pas des attributs des frères droits est dite L-attribuée.

Exemple

Soit une déclaration définie par la règle syntaxique suivante : D→TL {L.type=T.type}
Permet de passer l'information sur le type de la déclaration à la liste de variables L.

Exemple2 :

L→V,$L^{(1)}$ V.type=L.type;
$L^{(1)}$. type=L.type;
L→V $L^{(1)}$. type=L.type;

2.2 Graphe de dépendances

Si un attribut **b** à un nœud d'un arbre syntaxique dépend d'un attribut **c**. La règle sémantique définissant **b** en ce nœud doit être évaluée après la règle sémantique définissant c. les interdépendances entre les attributs aux nœuds d'un arbre syntaxique peuvent être décrites par un graphe orienté appelé Graphe de dépendance.

Donc, on appelle graphe de dépendances le graphe orienté représentant les indépendances entre les divers attributs. Le graphe a pour sommet chaque attribut. Il y a un arc de a à b si seulement si le calcul de b dépend de a.

Remarque

On construit le graphe de dépendances pour chaque règle de production, ou bien directement le graphe de dépendances d'un arbre syntaxique donné. C'est ce dernier qu permet de voir quel ordre évaluer les attribués.

Exemple 2

Soit un DDS qui fait intervenir deux attributs hérités **h** et **r** et deux attributs synthétisés **s** et **t**.

Production	**Action sémantique**
S→E	E.h:=2*E.s+1 S.r:=E.t
S→EE	E(0).s:=E(1).s+E(2).s E(0).t:=3*E(1).t-E(2).t E(1).h:=E(0).h E(2).h:=E(0).h+4
E→nombre	E.s:= nombre E.t:=E.h +1

Remarque

Si le graphe de dépendance contient un cycle, l'évaluation des attributs est alors impossible.

Exemple
Soit la DDS suivante (a hérité et b synthétisé)

Production	**Action sémantique**
S'→S	S.a:=S.b
S→SST	S(0).b:=S(1).b+S(2).b+T.b S(1).a:=S(0).a S(2).a:=S(0).a+1 T.a:=S(0).a+ S(0).b+S(1).a
T→PT	T(0).b:=P.a+P.b P.a:=T(0).a+3 T(1).a:=...

2.3 Construction des arbres abstraits

Un arbre syntaxique abstrait est une forme condensée d'arbre syntaxique, adapté à la représentation des constructions d'un langage.
Dans un arbre abstrait, les opérateurs et les mots clés n'apparaissent pas comme des feuilles, mais sont plutôt associées aux nœuds intérieurs qui seraient les pères de ces feuilles dans l'arbre syntaxique.
La traduction dirigée par la syntaxe peut se fonder sur des arbres abstraits de la même façon que sur les arbres syntaxiques.
La construction d'un arbre abstrait pour une expression est semblable à la traduction de l'expression en forme postfixé. On construit des sous arbres pour les expressions en créant un nœud pour chaque opérateur et chaque opérande. Les fils d'un nœud opérateur sont les racines des sous arbres représentant les expressions constituant les opérandes de cet opérateur.
Une DDS pour construire des arbres abstraits :

production	**action sémantique**
E→E+T	E.pn=CréerNoeud('+',$E^{(1)}$.pn,T.pn)
E→E-T	E.pn=CréerNoeud('-',$E^{(1)}$.pn,T.pn)
E→ T	E.pn=T.pn
T → (E)	T.pn= E.pn
T →id	T.pn=CréerFeuille(Id,ValLex)
T →nb	T.pn=CréerFeuille(nb,nb.ValLex)

3 Ordre d'évaluation

L'évaluation des attributs peut être faite selon deux approche, à, savoir ; après l'analyse syntaxique ou durant l'analyse syntaxique.

3.1 Après l'analyse syntaxique

On peut faire le calcul des attributs indépendamment de l'analyse syntaxique : lors de l'analyse syntaxique, on construit l'arbre syntaxique, puis ensuite, lorsque l'analyse syntaxique est terminée, le calcul des attributs s'effectue sur cet arbre par des parcours de cet arbre (en suivant l'ordre d'évaluation des attributs).
Cette méthode est très coûteuse en mémoire (stockage de l'arbre). Mais l'avantage est que l'on n'est pas dépendant de l'ordre de visite des sommets de l'arbre syntaxique imposé par l'analyse syntaxique, où l'analyse descendante impose un parcours en profondeur du haut vers le bas, de la gauche vers la droite, et l'analyse ascendante un parcours du bas vers le haut, ...).

Exemple : Avec un attribut synthétisé :

Évaluation d'une expression arithmétique avec une analyse ascendante

production	**action sémantique**	**traduction avec une pile**
E→E+T	E(0).val:=E(1).val+T.val	tmpT = depiler() tmpE = depiler() empiler(tmpE+tmpT)
E→ T	E.val:=T.val	
T→ T*F	T(0).val:=T(1).val*F.val	tmpT = depiler() tmpF = depiler() empiler(tmpT*tmpF)
T→ F	T.val:=F.val	
F→ (E)	F.val:=E.val	
F→ nb	F.val:=nb	empiler(nb)

L'analyse syntaxique s'effectue à partir de la table d'analyse LR (vue dans le chapitre précédent). Par exemple, pour le mot 2*(10+3) on obtiendra :

pile	entrée	action	pile des attributs
\$ 0	$2*(10+3)\$$	d5	
\$ 0 2 5	$*(10+3)\$$	r6 : $F \to$ nb	2
\$ 0 F 3	$*(10+3)\$$	r4 : $T \to F$	2
\$ 0 T 2	$*(10+3)\$$	d7	2
\$ 0 T 2 * 7	$(10+3)\$$	d4	2
\$ 0 T 2 * 7 (4	$10+3)\$$	d5	2
\$ 0 T 2 * 7 (4 10 5	$+3)\$$	r6 : $F \to$ nb	2 10
\$ 0 T 2 * 7 (4 F 3	$+3)\$$	r4 : $T \to F$	2 10
\$ 0 T 2 * 7 (4 T 2	$+3)\$$	r2 : $E \to T$	2 10
\$ 0 T 2 * 7 (4 E 8	$+3)\$$	d6	2 10
\$ 0 T 2 * 7 (4 E 8 + 6	$3)\$$	d5	2 10
\$ 0 T 2 * 7 (4 E 8 + 6 3 5	$)\$$	r6 : $F \to$ nb	2 10 3
\$ 0 T 2 * 7 (4 E 8 + 6 F 3	$)\$$	r4 : $T \to F$	2 10 3
\$ 0 T 2 * 7 (4 E 8 + 6 T 9	$)\$$	r1 : $E \to E+T$	2 13
\$ 0 T 2 * 7 (4 E 8	$)\$$	d11	2 13
\$ 0 T 2 * 7 (4 E 8) 11	\$	r5 : $F \to (E)$	2 13
\$ 0 T 2 * 7 F 10	\$	r3 : $T \to T*F$	26
\$ 0 T 2	\$	r2 : $E \to T$	26
\$ 0 E 1	\$	**ACCEPTÉ !!!**	26

3.2 Pendant l'analyse syntaxique

On peut évaluer les attributs en même tant que l'on effectue l'analyse syntaxique. Dans ce cas, on utilisera une pile pour conserver les valeurs des attributs, cette pile pouvant être la même que celle de l'analyseur syntaxique, ou une autre.
Cette fois-ci, l'ordre d'évaluation des attributs est lié à l'ordre dans lequel les noeuds de l'arbre syntaxique sont "crées" par la méthode d'analyse.

Remarque

On ne pourra traiter les grammaires S-attribuées qu'avec une analyse ascendante, et les grammaires L-attribuées qu'avec l'analyse descendante.
Exemple : avec un attribut hérité :
Compter le nombre de bit à 1 dans un mot binaire

Production	Action sémantique
S→B	B.nb=0
B→0B	$B^{(1)}$.nb=$B^{(0)}$.nb
B→1B	$B^{(1)}$.nb=$B^{(0)}$.nb+1
B→ε	Ecrire(B.nb)

On effectue une analyse descendante, donc il nous faut la table d'analyse LL…

Problème

Comment on fait quand on a à la fois des attributs synthétisés et hérités ?

Solution

Souvent, on peut utiliser des attributs synthétisés qui font la même chose que les hérités que l'on voulait.

Exemple

On peut modifier le DDS de l'exemple précédent pour être une DDS S-attribuée: comme suit:

Production	Action sémantique
S→B	Ecrire(B.nb)
B→0B	$B^{(0)}$.nb=$B^{(1)}$.nb
B→1B	$B^{(0)}$.nb=$B^{(1)}$.nb+1
B→ε	B.nb=0

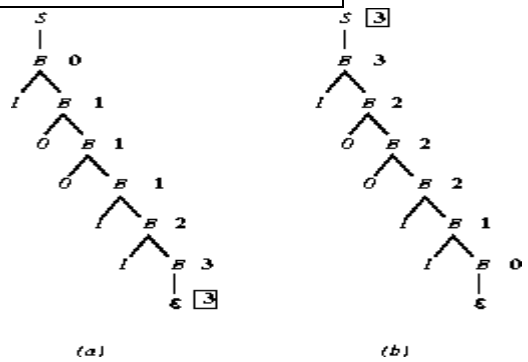


Figure 6.3 : L'arbre décoré pour 10011 avec des attributs (a)hérités (b) synthétisés

Parfois, il est nécessaire de modifier la grammaire. Par exemple, considérons une DDS de typage de variables dans une déclaration Pascal de variables (en Pascal, une déclaration de variable consiste en une liste d'identificateurs séparés par des virgules, suivie du caractère ':', suivie du type. Par Exemple a,b : integer)

production	action sémantique
D→L: T	L.type:=T.type
L → Id	mettre dans la table des symboles le type L.type pour Id
L → L, Id	$L^{(1)}$.type:=$L^{(0)}$.type mettre dans la table des symboles le type L(0).type pour Id
T → Integer	T.type:=entier
T→ Char	T.type:=caractere

C'est un attribut hérité. Pas possible de trouver un attribut synthétisé faisant la même chose avec cette grammaire. Changeons la grammaire !

production	action sémantique
D→id L	mettre dans la table des symboles le type L.type pour Id
L →, Id L	$L^{(0)}$.type:=$L^{(1)}$.type mettre dans la table des symboles le type L(1).type pour Id
L →:T	L.type:=T.type
T → Integer	T.type:=entier

T→ Char	T.type:=caractere

4 Schéma de traduction

Un schéma de traduction est une grammaire non contextuelle dans laquelle des attributs sont associés aux symboles de la grammaire et les actions sémantiques délimités par des accolades{ et}, sont insérées dans les parties droites des productions.

Exemple .

Un Schéma de traduction qui transforme des expressions infixées d'addition et de soustractions en expressions, postfixées équivalentes :

E→TR
R→'+'T{imprimer ('+')} R|ε
T→nb{ imprimer (nb)}

5 Portée des identificateurs

On appelle portée d'un identificateur les parties du programme où il est valide et a la signification qui lui a été donné lors de sa déclaration.

L'analyseur sémantique doit vérifier si chaque utilisation d'un identificateur est légale, si cet identificateur a été déclaré et cela de manière unique dans son bloc. Il faut donc mémoriser tous les symboles rencontrés au fur et à mesure de l'avancée dans le texte source. Pour cela on utilise une structure de données adéquate que l'on appelle une **table des symboles**.

La table des symboles contient toutes les informations nécessaires sur les symboles (variables, fonctions, ...) du programme :

- identificateur (le nom du symbole dans le programme)
- son type (entier, chaîne, fonction qui retourne un réel,...)
- son emplacement mémoire
- si c'est une fonction : la liste de ses paramètres et leurs types

Lors de la déclaration d'une variable, elle est stockée dans la table des symboles. Il faut préalablement regarder si cette variable n'est pas déjà contenue dans la table. Si c'est le cas, c'est une erreur. Lors de l'utilisation d'une variable, il faut vérifier qu'elle fait partie de la table (on peut alors récupérer son type, donner ou modifier une valeur, ...).

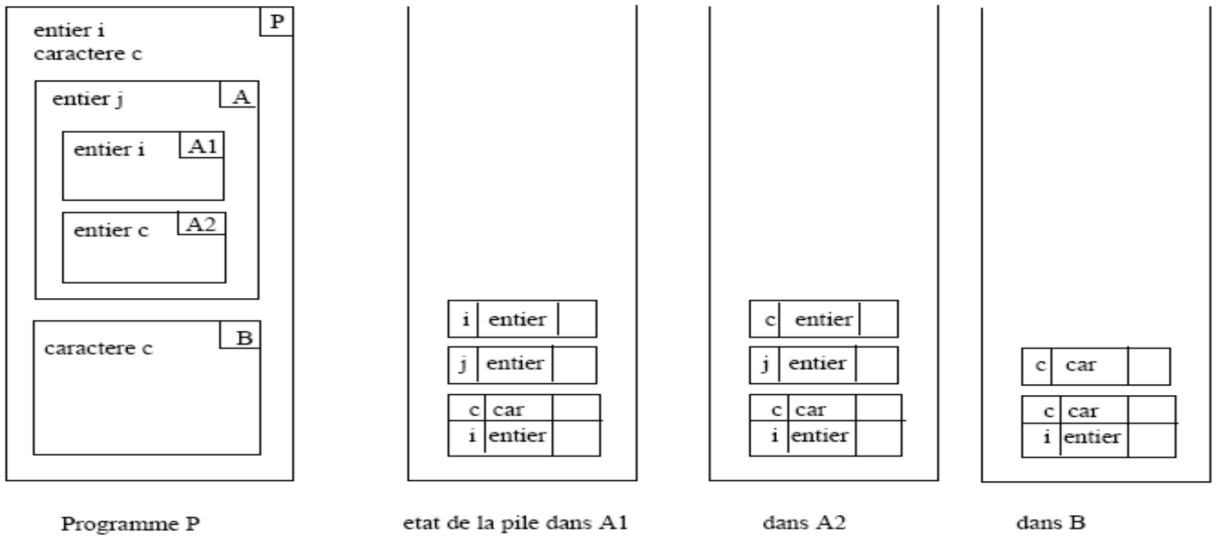


Figure 6.4: Exemple d'une pile d'exécution

6 Contrôle de type

Lorsque le langage source est un langage typé, il faut vérifier la pertinence des types des objets manipulés dans les phrases du langage. Par exemple en langage C, on ne peut pas additionner un double et un char *, multiplier un int et un struct, indicer un tableau avec un float, affecter un struct * à un char ... Certaines opérations sont par contre possibles : affecter un int à un double, un char à un int, ... (Conversions implicites).

On appelle statique un contrôle de type effectué lors de la compilation, et dynamique un contrôle de type effectué lors de l'exécution du programme cible.

Exemple

TDS de contrôle de type :

Règle de production	action sémantique	
$I \rightarrow Id = E$	I.type :=	si Id.type=E.type alors E.type sinon erreur_type_incompatible(ligne,Id.type,E.type)
$I \rightarrow$ si E alors I	$I^{(0)}$.type :=	si E.type=booleen alors $I^{(1)}$.type sinon erreur_type(ligne,...)
$E \rightarrow Id$	E.type :=	Recherche_Table(Id)
$E \rightarrow E + E$	$E^{(0)}$.type :=	si $E^{(1)}$.type=entier et $E^{(2)}$.type=entier alors entier sinon si ($E^{(1)}$.type≠reel et $E^{(1)}$.type≠entier) ou ($E^{(2)}$.type≠reel et $E^{(2)}$.type≠entier) alors erreur_type_incompatible(ligne,$E^{(1)}$.type,$E^{(2)}$.type) sinon reel
$E \rightarrow E$ mod E	$E^{(0)}$.type :=	si $E^{(1)}$.type=entier et $E^{(2)}$.type=entier alors entier sinon erreur_type(ligne,...)
	⋮	

Maintenant imaginons le boulot du compilateur C lorsqu'il se trouve confronté à un truc du genre

s→t.f(p[*i])-&j

Il faut alors trouver une représentation pratique pour les expressions de type.

7 Surcharge d'opérateurs et de fonctions

Des opérateurs (ou des fonctions) peuvent être surchargés c'est à dire avoir des significations différentes suivant le contexte. Par exemple, en C, la division est la division entière si les deux opérandes sont de type entier, et la division réelle sinon. Dans certains langages8.3, on peut redéfinir un opérateur standard, par exemple en Ada les instructions

function "*" (i,j : integer) return complexe;

function "*" (x,y : complexe) return complexe;

surchargent l'opérateur * pour effectuer aussi la multiplication de nombres complexes (le type complexe devant être défini). Maintenant, si l'on rencontre 3*5, doit-on considérer la multiplication standard qui retourne un entier ou celle qui retourne un complexe ? Pour répondre à cette question, il faut regarder comment est utilisée l'expression. Si c'est dans 2+(3*5) c'est un entier, si c'est dans z*(3*5) où z est un complexe, alors c'est un complexe. Bref, au lieu de remonter un seul type dans la

TDS, il faudra remonter un ensemble de types possibles, jusqu'à ce que l'on puisse conclure. Et je ne parle pas de la génération de code.

8 Fonctions polymorphes

Une fonction (ou un opérateur) est dite polymorphe lorsque l'on peut l'appliquer à des arguments de types variables (et non fixés à l'avance). Par exemple, l'opérateur & en C est polymorphe puisqu'il s'applique aussi bien à un entier qu'à un caractère. On peut également, dans certains langages, définir une fonction calculant la longueur d'une liste d'éléments, quelque soit le type de ces éléments.

Cette fois-ci, il ne s'agit plus seulement de vérifier l'équivalence de deux types, mais il faut les unifier.

Un des problèmes qui se pose alors est l'inférence de type, c'est à dire la détermination du type d'une construction du langage à partir de la façon dont elle est utilisée.

Ces problèmes ne sont pas simples et sont étudiés dans le cadre des recherches en logique combinatoire et en lambda-calcul. Le lambda-calcul permet de définir et d'appliquer les fonctions sans s'occuper des types.

Références

Références

1. Alfred Aho - Monica Lam- Ravi Sethi- Jeffrey Ullman Compilers: principles, techniques, and tools. Vol. 2. Reading: Addison-wesley, 2nd Edition , 2007.
2. Doug Brown, John Levine, Tony Mason - Lex and Yacc [parsers, UNIX-O'Reilly Media (1995)
3. Andrew W. Appel, Maia Ginsburg - Modern Compiler Implementation in C-Cambridge University Press (1998)
4. Henri Garreta, Techniques et outils pour la compilation, Faculté des Sciences de Luminy - Universit´e de la Méditerranée (support des cours), Janvier 2001