Compilation

Retima Farida

Février 2024

Table des matières

Compilation (Introduction)

1. Introduction

En général, les programmeurs rédigent un programme informatique en utilisant un langage de programmation de haut niveau. Mais un ordinateur ne comprend pas le langage haut niveau. Il ne comprend que le **code machine** (les programmes en binaire 0 et 1). Un code source désigne un programme écrit dans un langage évolué. Le code source doit être converti en code machine, ce qui est accompli par les compilateurs et les interpréteurs. Un compilateur ou un interpréteur est donc un logiciel qui transforme un programme écrit en langage évolué en un code machine qui est compris par l'ordinateur.

2. Objectif du cours

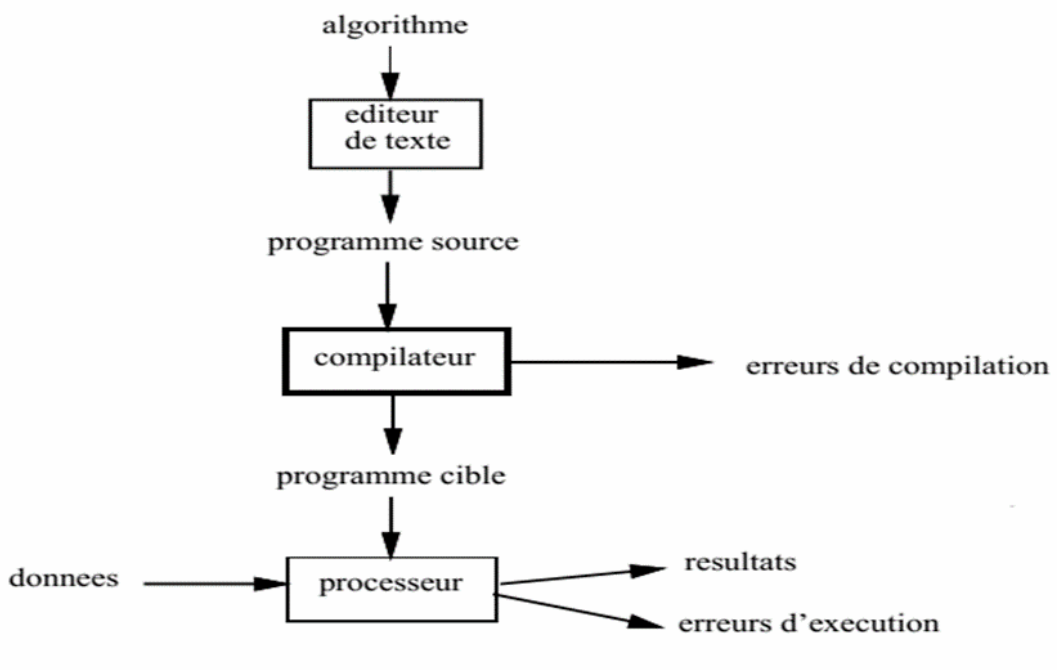
Fondamental

- Compréhension du cheminement d'un programme source vers un programme en langage machine.
- Analyse des différentes phases du processus de compilation d'un langage évolué.
- Les fondamentaux de la création de compilateurs comprennent l'analyse lexicale, l'analyse syntaxique, l'analyse sémantique et la génération de code.
- les principaux outils utilisés pour réaliser ces analyses : théorie des langages (expressions régulières, automates, grammaires).
- Apprendre à utiliser des outils de génération d'analyseurs lexicaux et syntaxiques (LEX et YACC).

3. Définition d'un compilateur

Définition

- Il est impossible pour un ordinateur d'exécuter un programme écrit dans un langage de haut niveau s'il n'est pas traduit en instructions exécutables par l'ordinateur (instructions élémentaires directement exécutables par le processeur).
- Un compilateur est donc un programme spécifique qui convertit un programme écrit dans un langage de haut niveau (par le programmeur) en instructions exécutables par un la machine.



chaîne de développement d'un programme

4. Structure d'un compilateur

4.1. Introduction

La compilation se décompose en deux phases:

1. **Une phase d'analyse**: consiste à identifier les variables, les instructions, les opérateurs et à vérifier la structure syntaxique du programme ainsi que certaines caractéristiques sémantiques. Trois parties composent cette phase (**analyse lexicale, analyse syntaxique, analyse sémantique**).
2. **Une phase de production**: permet de générer le code cible. Il est constitué de deux parties distinctes : **la génération de code et l'optimisation de code**.

4.2. La phase d'analyse

a) Analyse lexicale

Définition

Le rôle principal de l'analyseur lexical est de lire le texte du code source (une suite de caractères), puis de créer **des unités lexicales** (connues également sous le nom d'entités lexicales, de lexèmes, de jetons, de tokens ou encore d'atomes lexicaux).

Exemple

Par exemple à partir du morceau de C suivant:

If (i<a+b) x=x*2;

L'analyseur lexical déterminera la suite de token:

Les tokens	Signification
if	Mot clé
(	séparateur

i	identificateur
<	opérateur relationnel
a	identificateur
+	opérateur arithmétique
b	identificateur
)	séparateur
x	identificateur
=	affectation
x	identificateur
*	opérateur arithmétique
2	constant
;	séparateur

b) Analyse syntaxique

Définition

Dans l'analyse syntaxique, l'ordre des tokens est vérifié pour être conforme à l'ordre défini pour le langage. La syntaxe du langage est vérifiée à partir de la définition de sa grammaire. L'analyseur syntaxique sait comment doivent être construites les expressions, les instructions, les déclarations de variables, les appels de fonctions…

En d'autres mots, le rôle principal de l'analyseur syntaxique consiste à vérifier la syntaxe du code en regroupant les unités lexicales selon des structures grammaticales qui permettent de créer une représentation syntaxique du code source. La structure de cette dernière est souvent en arbre.

Exemple

En C, un exemple d'une instruction doit se présenter sous la forme:

if (expression) instruction

Si l'analyseur syntaxique reçoit la suite d'unités lexicales :

Mot clé identificateur opérateur relationnel constant séparateur identificateur affectation identificateur opérateur arithmétique identificateur séparateur

il doit signaler que ce n'est pas correct car il n'y a pas de **(** juste après le if.

Mais si l'analyseur syntaxique reçoit la suite d'unités lexicales :

mot clé séparateur identificateur opérateur relationnel constant séparateur identificateur affectation identificateur opérateur arithmétique constant séparateur

L'analyseur syntaxique signale que cette suite des unités lexicales est syntaxiquement correcte.

c) Analyse sémantique ou vérification de type

Définition

À cette étape, on s'assure que les variables ont un type correct. À titre d'exemple, il est nécessaire de vérifier que la variable « i » est bien d'un type « entier » et que la variable « a » est bien un nombre. De plus, il est impossible, par exemple : multiplier un réel avec une chaîne de caractères, ou assigner une variable à un nombre. ….

Exemple

Dans l'analyse sémantique de « **a := b + 2 * c ; »,** il faut vérifier que, si a est de type entier, alors b et c le sont aussi, sinon il faut indiquer une erreur.

4.3. La phase de production

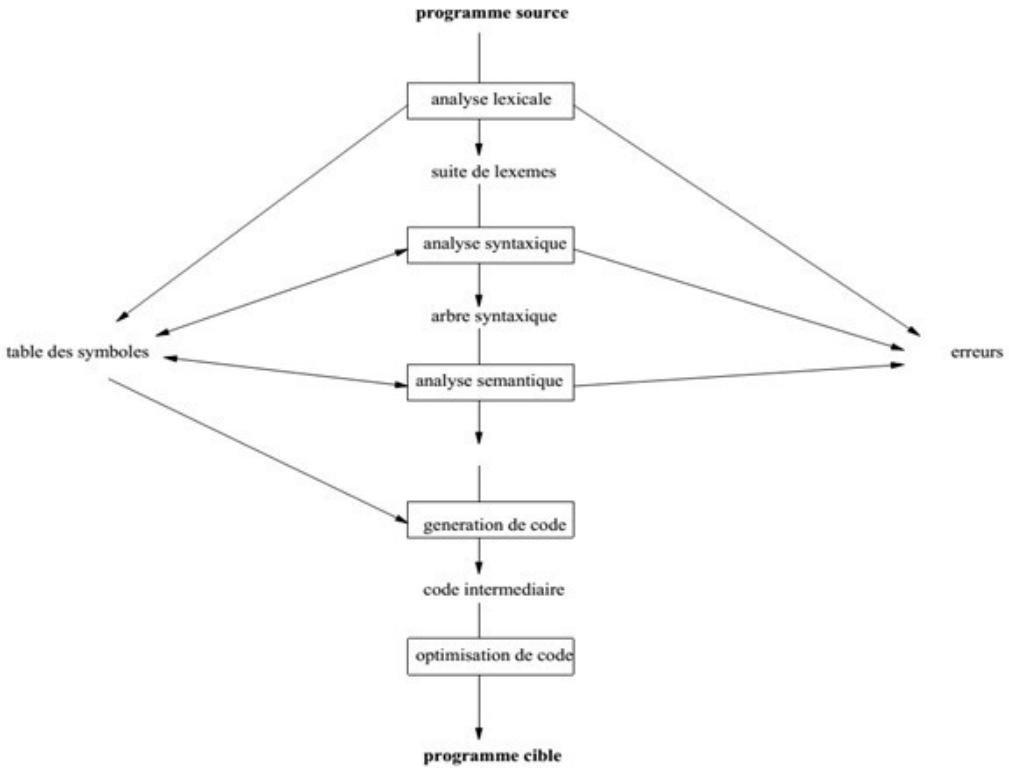
Méthode

1) Génération de code : Le générateur de code représente la dernière étape du compilateur. Son objectif est de fournir une représentation intermédiaire du programme source et de produire un programme cible équivalent, qui doit être correct et de bonne qualité, et doit également être exécuté le plus rapidement possible.

2) Optimisation de code : L'objectif est d'améliorer le code produit pour rendre le programme final plus rapide.

Il y a des optimisations qui ne dépendent pas de la machine cible : La suppression des calculs superflus, propagation des constantes, extraction des boucles des invariants de boucle.

Et il y a des optimisations qui dépendent de la machine cible : remplacer des instructions générales par des instructions plus efficaces et plus adaptées, utilisation optimale des registres.



Structure d'un compilateur

5. Interpréteur versus Compilateur

Remarque

Un interpréteur : est un logiciel qui est responsable de l'interprétation du programme source lors de son exécution. En d'autres termes, il examine les instructions du programme source successivement et les exécute immédiatement. Dans cette situation, il n'existe pas de programme objet équivalent.

Un compilateur :Il s'agit d'un logiciel informatique qui convertit tout le code source d'un projet logiciel en code machine avant de l'exécuter. Ce n'est qu'après cette traduction que le projet sera exécuté par le processeur qui a toutes les instructions sous forme de code machine.

interpréteur	**compilateur**
Convertit le programme successivement (ligne après l'autre)	convertir tout le code
Aucun code d'objet n'est généré	Génère du code d'objet
Continuer à traduire le programme jusqu'à ce que la première erreur soit détectée.	Le message d'erreur ne se produit qu'après avoir examiné l'intégralité du programme (de façon groupée à la fin de la compilation)
La traduction du code source est pendant le fonctionnement du logiciel	La traduction du code source est avant l'exécution du logiciel
Exemple d'interpréteur : PHP, Python….	Exemple de compilateur : C, C++

Compilateur Vs Interpréteur

6. Exercice (Questions de cours)

6.1. Exercice

1. C'est quoi un compilateur ?
2. Donnez la structure logique d'un compilateur ?
3. Donner un exemple d'une erreur, qui peut être détectée, au niveau de l'analyse syntaxique et sémantique ?
4. Quelle est la différence entre la compilation et l'interprétation ?

Analyse lexicale

1. Introduction

- Le processus d'analyse lexical est la première étape d'un compilateur. Son rôle principal consiste à lire les caractères d'entrée et à générer une série **d'unités lexicales** que l'analyseur syntaxique devra traiter.
- De plus, l'analyseur lexical peut également effectuer certaines tâches supplémentaires, dont **l'élimination des commentaires** et des **espaces** qui apparaissent dans le programme source sous forme de caractères blancs, de **tabulation** ou de **fin de ligne**. De plus, il gère les **numéros de ligne** dans le programme source afin de pouvoir lier chaque erreur rencontrée au numéro de ligne correspondant.

2. Unité lexicale

Définition

Une unité lexicale est une suite de caractères qui a une signification collective

Exemple

Les chaines <,>,= sont des opérateurs relationnels, **l'unité lexicale** est **OPREL** par exemple

Définition

Un **modèle** correspond à une **règle** qui décrit **toutes les chaînes** du programme qui peuvent correspondre à cette **unité lexic**

Définition

La suite de caractères du programme source qui correspond au modèle d'une unité lexicale est appelée **lexème**.

Exemple

- L'unité lexicale IDENT (identificateur) en C a pour modèle toute suite non vide de caractère composé de chiffre, lettre, ou des symboles et qui commence par une lettre
- Exemple de lexème pour cette unité lexicale sont a ; b ; somme, ….
- L'unité lexicale NOMBRE (entier signé) a pour modèle : toute suite non vide de chiffres précédée éventuellement d'un seul caractère {+,-}. Lexèmes possibles : -10, 56
- Pour décrire un **modèle** d'une unité lexicale on utilisera **les expressions régulières.**

3. Expression régulière

Rappel

Une expression régulière est une **formule** qui désigne un ensemble de chaînes de caractères composées d'un alphabet ∑. Cette collection de chaînes de caractères est désignée comme un **langage**.

- Une expression régulière est une notation pour décrire un langage régulier.
- On appelle **alphabet** un ensemble fini non vide ∑ de symboles.
- On appelle **mot** toute séquence finie d'éléments de ∑.
- On note ε le mot vide.
- On note ∑* ensemble infini contenant tous les mots possibles sur ∑.
- On note ∑+ ensemble des mots non vides que l'on peut composer sur ∑, c'est-à-dire ∑+=∑*-{ε}
- On note| m| la longueur du mot m c'est-à-dire le nombre de symboles de ∑ composant le mot.

Exemple

1- Soit l'alphabet ∑= {a,b,n}

aab, bbbanb, n, ε, na sont des mots de ∑*.

2 - Soit C={a,b}

- L'expression régulière a |b indique l'ensemble {a,b}
- L'expression régulière (a|b) (a|b) indique {aa,ab,ba,bb} l'ensemble de toutes les chaines de a et b de longueur 2.
- L'expression régulière a* indique l'ensemble de toutes les chaines formées d'un nombre quelconque de a : c. à. d {ε,a,aa,aaa, ……...}
- On peut décrire un identificateur comme : Identificateur = lettre (lettre | chiffre|sep)* tel que : lettre= [a-zA-Z] Chiffre=[0-9] Sep=_.

Définition

On appelle **langage** sur un alphabet ∑ tout sous ensemble de ∑*.

Exemple

Soit l'alphabet ∑={a,b}

Soit L l'ensemble des mots de ∑* ayant autant de a que de b. L est le langage infini { ε,ab,bb,…}.

Définition

un langage régulier L sur un alphabet ∑ est défini récursivement de la manière suivante :

- {ε} est un langage régulier sur ∑
- Si b est une lettre de ∑, {b} est un langage régulier sur ∑.
- Si F est un langage régulier sur ∑, alors F^n et F^* sont des langages réguliers sur ∑.
- Si F1 et F2 sont des langages réguliers sur ∑ , alors F1U F2 et F1F2 sont des langages réguliers.

Les langages réguliers se décrivent par une expression régulière (ER).

Définition

Définitions régulières

On appelle une définition régulière la nomination des expressions régulières. Ces noms serviront à élaborer 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 ER.

Exemple

lettre → A | B | . . . | Z | a | b | . . . | z

chiffre → 0 | 1 | . . . | 9

ident → lettre (lettre | chiffre)*

chiffres → chiffre chiffre *

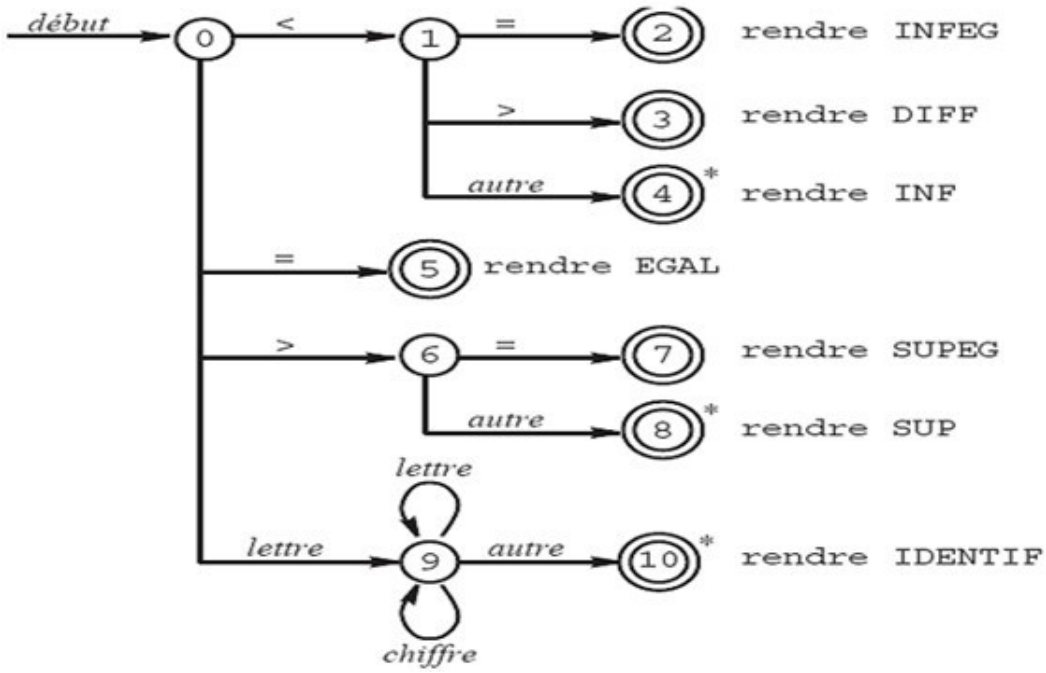
4. Reconnaissance des unités lexicales

Définition

Un reconnaisseur pour un langage est un programme qui reçoit une chaîne x et répond oui si x est un mot du langage, et non autrement. Une expression régulière est compilée en un reconnaisseur en utilisant un diagramme de transition généralisé appelé **automate fini.**

Exemple

- Pour illustrer cette définition :le problème de la reconnaissance des unités lexicales INFEG, DIFF, INF, EGAL, SUPEG, SUP, IDENTIF, respectivement définies par les expressions régulières <=, <>, <, =, >=, > et lettre(lettre/chiffre)*.
- La figure suivante montre les diagrammes traduisant la reconnaissance des unités lexicales INFEG, DIFF, INF, EGAL, SUPEG, SUP et IDENTIF.



reconnaissance des unités lexicales INFEG, DIFF, INF, EGAL, SUPEG, SUP et IDENTIF.

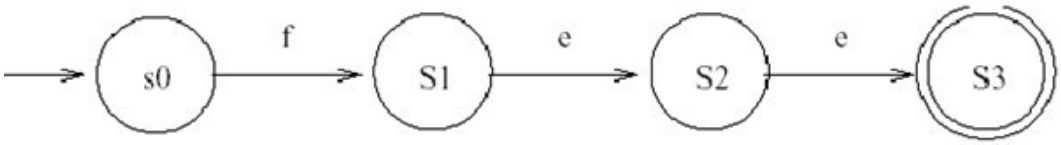
Rappel

Présentation de l'automate :

La représentation d'un automate est généralement illustrée par un graphe orienté avec les sommets sont les états et les arêtes étiquetées correspondent aux transitions. On distingue les états d'acceptation que l'on entourera d'un cercle, ainsi que l'état initial.

Exemple

Sur ce dessin, l'état initial est S0, S3 est l'état final signifiant qu'on a reconnut le mot « fee ».Lorsque nous lisons un autre mot, une des transitions ne sera pas possible, nous rejetterons le mot.



un automate

Remarque

Un **AFN** (Automate Fini Non déterministe) définit le langage comme étant l'ensemble des chaînes d'entrée qu'il **accepte**.

Un AFN accepte une chaîne d'entrée x tant qu'il y a un chemin spécifique dans le graphe de transition entre l'état initial et l'état final.

Les automates fini peut être déterministes ou non :

- **Les automates finis non déterministe (AFND):**

  Un automate fini non déterministe (AFND) est un quintuplet A=(Q , ∑, δ, q0, F), où

  - Q est un ensemble fini d'états
  - ∑ est un alphabet

- δ : est la fonction de transition
- q0 est l'état initial
- F est l'ensemble des états terminaux ou finaux.

Un automate non-déterministe est un automate qui présente la particularité suivante :

- Plusieurs arcs reconnaissant le même caractère peuvent « sortir » du même état.
- Il peut y avoir des transitions, notées ε, qui ne correspondent à une aucune reconnaissance de caractères.
- Le langage reconnu par un automate (Q , ∑,δ,q0, F) est l'ensemble des mots w tels qu' il existe : q0-->wF.

- **Automate fini déterministe (AFD)**

  L'automate est dit déterministe lorsque :

  1. la fonction δ associe à chaque couple (état, lettre) un état unique
  2. aucun état n'a de ε transition

Remarque

Le passage d'un **AEFND** vers un **AEFD** a pour but :

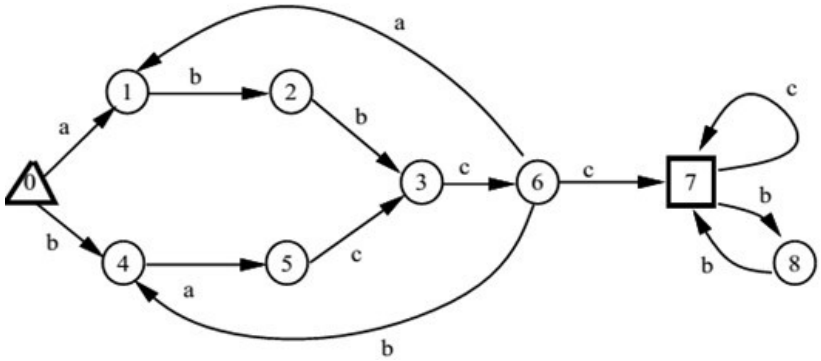
Le processus de reconnaissance d'un mot (par exemple : analyse lexicale) par un AEFND est extrêmement complexe. Ainsi, cela a un impact direct sur le temps de reconnaissance d'un mot. C'est pourquoi il est important de transformer l'AEFND en AEFD.

5. Analyseur lexical

- Le rôle d'un analyseur lexical est la **reconnaissance des unités lexicales**. Une unité lexicale peut être exprimée sous forme de définitions régulières.
- Dans la théorie des langages on définit des **automates** qui sont permettant la reconnaissance de mots. Un langage régulier est reconnu par un automate fini.

Exemple

Le langage régulier décrit par l'ER $(abbc|bacc)^+ c(c|bb)^*$ est reconnu par l'automate :



Méthode

Un langage régulier et donc une ER peut être reconnu par un **automate**. Ainsi, afin de créer un analyseur lexical de notre programme source, il est simplement nécessaire de créer un programme **qui simule l'automate**. Une fois qu'une unité lexicale est reconnue, elle est transmise à l'analyseur

syntaxique qui la traite, puis repasse la main à l'analyseur lexical qui lit l'unité lexicale suivante dans le programme source. Et ainsi de suite, jusqu'à ce que l'on rencontre une erreur ou jusqu'à ce que le programme source soit complètement traité.

6. Générateur d'analyseur lexical : LEX

Un Générateur d'analyseur lexical vise à :

- Prend en entrée la définition des unités lexicales.
- Produit un automate fini déterministe permettant de reconnaitre les unités lexicales.
- L'automate est produit sous la forme d'un programme C.
- Lex accepte en entrée des spécifications d'unités lexicales sous forme de définitions régulières et produit un programme écrit dans un langage de haut niveau (le langage C) qui une fois compilé, reconnait ces unités lexicales (ce programme est donc un analyseur Lexical)

Fondamental

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

%{

Partie 1 : déclarations pour le compilateur C

%}

Partie 2 : définitions régulières

%%

Partie 3 : règles

%%

Partie 4 : fonctions C supplémentaires

Définition

Partie 1 : les déclarations

Cette partie d'un fichier Lex peut contenir :

Un code écrit dans le langage cible, encadré par **%{ et %}**, qui se retrouvera au début du fichier synthétisé par Lex. C'est ici que l'on va spécifier les fichiers à inclure. Lex recopie tel quel tout ce qui est écrit entre ces deux signes, qui devront toujours être placés en début de ligne.

La partie 2 : Des définitions régulières définissant des notions non terminales, telles que lettre, chiffre et nombre. Ces spécifications sont de la forme :

notion expression régulière

Partie 3 : les règles

Cette partie sert à indiquer à Lex ce qu'il devra faire lorsqu'il rencontrera telle ou telle notion. Celle-ci peut contenir :

Des productions de la forme : ***expression régulière action***

où l' expression Régulière est écrit au début de la ligne ; action est un morceau de code source C, qui sera recopié tel quel, au bon endroit, dans la fonction **yylex().**

La fonction **yylex()** qui doit être appelée pour utiliser l'analyseur lexical :

- analyse séquentiellement un fichier d'entrée
- retourne 0 lorsqu'elle rencontre une fin de fichier
- effectue des opérations spécifiées par le programme Lex, lorsqu'une unité lexicale est reconnue

Enfin, la variable **`yytext`** désigne dans les actions les caractères acceptés par expression régulière. Il s'agit d'un tableau de caractère de longueur **`yyleng`**.

Partie 4 : fonctions C supplémentaires

On peut mettre dans cette partie facultative tous le code que tu veux. Si tu ne mets rien, Lex considère que c'est juste :

```
main()
{
yylex();
}
```

Exemple

```
%{
 #include <stdio.h>
#include <stdlib.h>
%}
/* Expressions regulieres */
chiffre    [0,9]
lettre     [a-z A-Z]
entier    {chiffre}+
ident     {lettre}({lettre}|{chiffre})*
%%
«:= » {return AFF ;}
« <= » {return OPREL ;}
if|IF|If|iF   {return MC_IF ;}
{entier}     {return NB ;}
{ident}     {return ID ;}
%%
```

Remarque

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 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}
r_1r_2	r_1 suivie de r_2	ab
$r_1\|r_2$	r_1 ou r_2	c|d
r_1/r_2	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 20

Les expressions réguliers Lex

7. Exercices

7.1. Exercice 01

Soit l'alphabet A= {a, b}.

1. Définir l'expression régulière R correspondant au langage régulier :
2. L= $\{a^n b^m\}$ où n>=0, m>0
3. Définir l'expression régulière pour l'ensemble de nombres entiers (signés ou non) sur l'alphabet A= {+,-,0,1,…9}.
4. Définir l'expression régulière pour l'ensemble des mots sur A={a,b} terminant par aba.
5. Soit l'alphabet A = {0,1}. Définir les expressions régulières correspondant aux langages ci-dessous :
   - Toutes les chaînes qui se terminent par 00.
   - Toutes les chaînes dont le 10ème symbole, compté à partir de la fin de la chaîne, est un 1.
6. Définir l'expression régulière pour le langage contenant tous les mots « begin » écrits avec des majuscules et minuscules mélangés : bEgin, BEGin, begiN, BEGIN, …

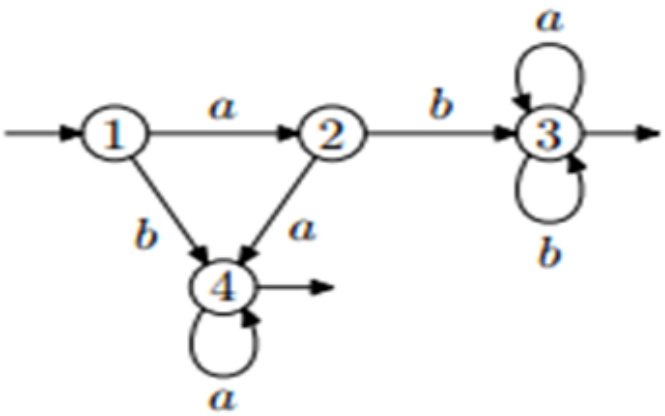
7.2. Exercice 02

1. Donner tous les mots de tailles 0, 1, 2, 3, et 4 des langages réguliers suivants :

   (a|ba)*

   a(aa |b(ab)*a)*a

2. Donner tous les mots de longueur 0, 1, 2, 3 et 4 reconnus par l'automate suivant :



7.3. Exercice 03

Construire les automates reconnaissants les langages suivant:

- Le langage des mots contenant au moins une fois la lettre a
- Le langage des mots contenant au plus une fois la lettre a
- Le langage des mots contenant un nombre pair de fois la lettre a

7.4. Exercice 04

Soit la table de transition de l'automate AEF **A** :

L'état initial est 0, et l'unique état final est 3.

a) Construire un automate à états finis **A** à partir de ce table de transition.

b) Construire un automate à états finis **déterministe** (équivalent à A).

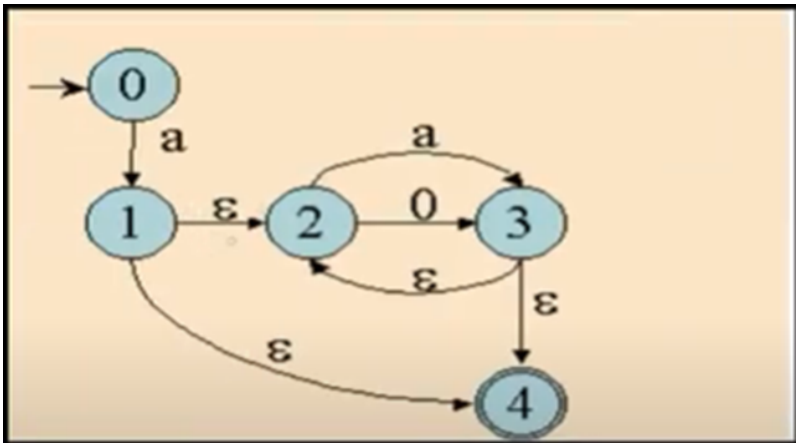
Etats	Symbole d'entrée	
	a	b
0	1, 2	
1	1, 2, 3	2, 3, 0
2	2	
3	0, 2	

Table de transition A

7.5. Exercice 05

Soit l'automate AEFND A avec ε-transition suivant :

1) Transformer l'automate **A** à un automate à états finis **déterministe**.



Automate AEFND « A »

7.6. Exercice 06

On considère le programme en C suivant :

```
Main ()
{//ceci est un programme
int a,b,c ;
a=7 ;
for(int i=0 ;i<a ;i++)
{
b=b+i*i ;
c=c*b ;
}
}
```

1- Indiquer sur ce programme comment fonctionner l'analyseur lexicale et quelles sont toutes les unités lexicales contenues ?

Analyse syntaxique

1. Introduction

L'analyse syntaxique est principalement utilisée pour vérifier si l'écriture du programme source conforme avec la syntaxe du langage à compiler. On peut définir cette dernière en utilisant une **grammaire hors contexte**.

L'analyseur syntaxique reçoit une série d'unités lexicales par l'analyseur lexical et doit vérifier que cette série peut être créée par la grammaire du langage.

Le problème est donc :

- on a la grammaire G
- et le mot m (un programme)

Est ce que m appartient au langage génère par G ? Le principe est d'essayer de construire un **arbre de dérivation.**

Il existe plusieurs méthodes d'analyse :l'analyse **descendante** et l'analyse **ascendante**.

Dans l'analyse **descendante** nous essayons de dériver à partir de l'axiome de la grammaire le programme source. Et l'analyse **ascendante** établit des réductions sur les chaines à analyser pour obtenir l'axiome de la grammaire.

2. Grammaire et arbre de dérivation

2.1. Grammaire

Définition

Une grammaire **hors contexte** est un quadruplet :

G = (T, NT, S, P) où :

- **T** est l'ensemble des **symboles terminaux** du langage. Les symboles terminaux sont les mots identifiés par l'analyseur lexical « unité lexicale ».par exemple : if , else sont des terminaux.
- **NT** est l'ensemble des **symboles non-terminaux** du langage. Ce sont des symboles qui ne se trouvent pas dans le langage, mais plutôt dans les règles de la grammaire. Ils offrent la possibilité de décrire la structure des règles grammaticales.
- **S** ϵ **NT** est appelé l'élément de départ de G (ou **axiome de G**). Le langage décrit par G (noté L(G)) correspond à toutes les phrases qui peuvent être dérivées à partir de S selon les règles de la grammaire..
- **P** est un ensemble de **production** de la forme N → α1 α2… αn avec αi ϵ (TUNT)*. C'est à dire que chaque élément de P associe un non terminal à une suite de terminaux et non terminaux. Le fait que les parties gauches des règles ne comportent qu'un seul non terminal **donne la propriété "hors contexte" à la grammaire.**

Exemple

Symboles terminaux : T= {a,b}

Symboles non terminaux NT={S}

Axiome=S

Règles de production : {S →ε S →aSb}

2.2. Arbre de dérivation

Définition

On appelle dérivation **l'application d'une ou plusieurs règles** à partir d'un mot de(T U NT)+ On notera → une dérivation obtenue par application d'une seule règle de production, et → * une dérivation obtenue par l'application de n règles de production où n > 0.

Remarque

On peut produire des phrases qui sont syntaxiquement correctes mais qui n'ont pas de sens.Il sera possible de résoudre ce problème en utilisant l'analyse sémantique.

Définition

En considérant une grammaire G, on écrit L(G) le langage produit par G et défini par tous les mots constitués uniquement de symboles terminaux que l'on peut former à partir de S.

Définition

Un arbre de dérivation syntaxique pour la grammaire G de racine Sϵ NT et de feuilles wϵT* est un arbre ordonné dont la racine est S, les feuilles sont étiquetées par des terminaux formant le mot w.

Dérivations gauches et droites :

- Dérivation gauche consiste à réécrire le symbole non-terminal le plus à gauche à chaque étape.
- Dérivation droite consiste à réécrire le symbole non-terminal le plus à droite à chaque étape.

Exemple

Soit la grammaire ayant S pour axiome et pour règles de production

P ={ S→ aT b |c

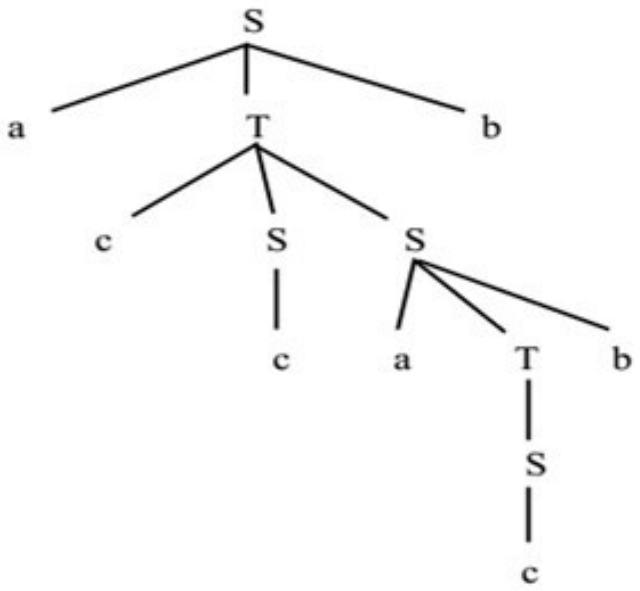
T→ cSS |S}

Un arbre de dérivation pour le mot accacbb est :

S→ aTb→ acSSb→ accSb→ accaTbb→ accaSbb→ accacbb **(dérivation gauche)**

Ou S→ aTb → acSSb→ acSaTbb→ acSaSbb→ acSacbb→ accacbb **(dérivation droite)**

Ces deux suites différentes de dérivations donnent **le même arbre de dérivation**



Arbre de dérivation

3. Grammaire LL(1)

3.1. Introduction

Une grammaire **ambiguë** ou **récursive à gauche** ou **non factorisée à gauche** n'est pas LL(1).

3.2. Ambiguïté

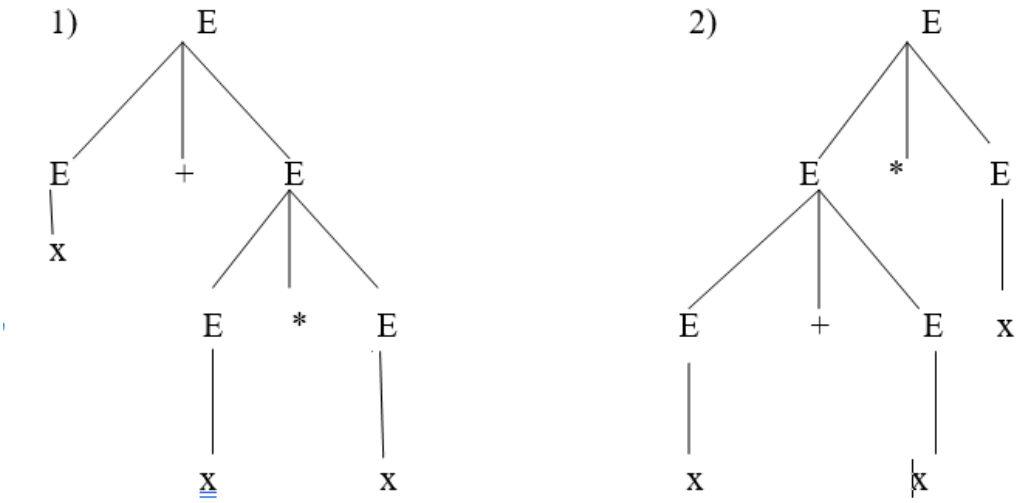
Définition

Si une phrase d'un langage L(G) possède deux arbres de dérivations distincts, alors cette phrase est ambiguë et la grammaire associée est elle-aussi ambiguë.

Exemple

La grammaire de l'expression suivantes est ambiguë ; il y'a deux arbres syntaxiques possibles pour x+x*x

E→ E+E |E*E |x



*les Arbres syntaxique pour x+x*x*

3.3. Récursivité à gauche

Définition

Une grammaire **G** est **Récursive Gauche** (**RG**) s'il existe une dérivation de la forme : $A \rightarrow^{+} Aw$, avec $A \in N$ et $w \in (NUT)^*$ ou N (ensemble de symboles non terminaux), T (symboles terminaux)

Élimination de la récursivité à gauche immédiate

Méthode

Remplacer toute règle de la forme $A \rightarrow A\alpha | \beta$ par les deux règles :
$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' | \epsilon$
Ou : toute règle de la forme $A \rightarrow A\alpha_1 | A\alpha_2 | .. A\alpha_n | \beta_1 | \ldots \beta_n$ par les deux règles:
$A \rightarrow \beta_1 A' | \ldots \beta_n A'$
$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \ldots \alpha_n A' | \epsilon$

Élimination de la récursivité à gauche immédiate

Élimination de la récursivité à gauche non immédiate

Méthode

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 $A_i \rightarrow A_j\alpha$ où $A_j \rightarrow \beta_1 | \ldots | \beta_p$ par $A_j \rightarrow \beta_1\alpha | \ldots | \beta_p\alpha$
  fin pour
  éliminer les récursivités à gauche immédiates des productions A_i
fin pour

Élimination de la récursivité à gauche non immédiate

Exemple

Soit la production suivante:

E→E+T|C

Donc pour éliminer la récursivité à gauche, on obtient :

E→CE'

E'→+TE'| Є

3.4. Factorisation à gauche

Définition

L'idée principale est que lorsque l'on développe un non-terminal A et qu'il n'est pas évident de décider quelle production utiliser, il est nécessaire de réécrire les productions de A de manière à différer la décision jusqu'à ce que suffisamment de texte soit lu pour faire le bon choix.

Exemple

$$\begin{cases} S \rightarrow bacdAbd|bacdBcca \\ A \rightarrow aD \\ B \rightarrow cE \\ C \rightarrow \dots \\ E \rightarrow \dots \end{cases}$$

Initialement, afin de déterminer si le choix est entre la première production ou la deuxième, il est nécessaire de lire la cinquième lettre du mot (un a ou un c). Il est donc impossible de déterminer dès le départ quelle production prendre.

Méthode

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

Fin pour
Recommencer jusqu'à ne plus en trouver.

algorithme de la factorisation à gauche

Exemple

E→ abE|abA|abC

Factorisée à gauche, cette production devient :

E→ abE'

E'→ E|A|C

4. Mise en œuvre d'un analyseur syntaxique

4.1. Introduction

L'analyseur lexical fournit à l'analyseur syntaxique **une série d'unités lexicales** (de symboles terminaux). Il est nécessaire de déterminer si cette suite (le mot) est **syntaxiquement correcte**, c'est-à-dire si c'est un mot du langage généré par sa grammaire. Il doit donc tenter de développer **l'arbre de dérivation** de ce mot. Si cela se réalise, le mot est syntaxiquement correct, sinon il est incorrecte.

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

4.2. Analyse descendante

a) Introduction

Selon l'analyse descendante, l'arbre est construit en descendant de la racine (c'est-à-dire l'axiome de départ) vers les feuilles (les unités lexicales). L'objectif est de déterminer à chaque étape quelle est la règle qui permet de créer le mot que nous lisons.

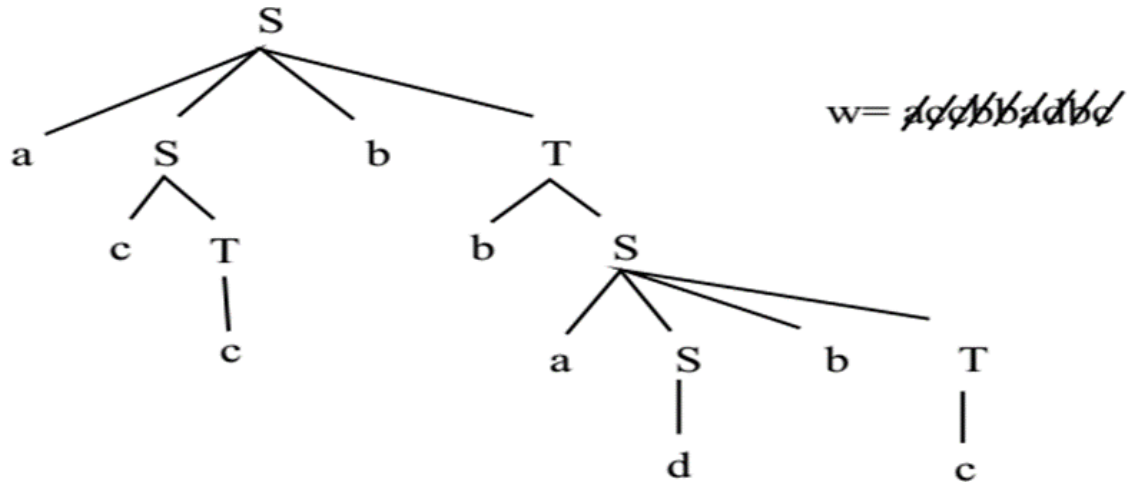
S→ aSbT|cT|d avec le mot w=accbbadbc

T→ aT|bS|c

On part avec l'arbre contenant le seul sommet S

La lecture de la première lettre du mot (a) nous permet d'avancer la construction comme l'indiquer la figure suivante.

→ donc on a trouvé un arbre de dérivation, donc le mot appartient au langage.



Remarque : Si notre grammaire est LL(l), l'analyse syntaxique peut se faire par l'analyse descendante. Mais comment savoir que notre **grammaire est LL(1)** ?

- Étant donnée une grammaire
- La rendre non ambiguë
- Eliminer la récursivité à gauche si nécessaire
- La factoriser à gauche si nécessaire
- Construire la table d'analyse

b) Table d'analyse LL(1)

i) Introduction

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

ii) Calcul de PREMIER :

Pour toute chaine composée de symboles terminaux et non_terminaux, on cherche PREMIER(α) l'ensemble de tous les terminaux qui peuvent commencer une chaine qui se dérive de α.

Exemple

On la grammaire suivant :

$$\begin{cases} E \to TE' \\ E' \to +TE' \mid -TE' \mid \varepsilon \\ T \to FT' \\ T' \to *FT' \mid /FT' \mid \varepsilon \\ F \to (E) \mid \text{nb} \end{cases}$$

$$\begin{aligned} \text{PREMIER}(E) &= \text{PREMIER}(T) = \{(, \text{nb}\} \\ \text{PREMIER}(E') &= \{+, -, \varepsilon\} \\ \text{PREMIER}(T) &= \text{PREMIER}(F) = \{(, \text{nb}\} \\ \text{PREMIER}(T') &= \{*, /, \varepsilon\} \\ \text{PREMIER}(F) &= \{(, \text{nb}\} \end{aligned}$$

iii) Calcul de SUIVANT

Pour tout non_terminal A, on cherche SUIVANT(A), l'ensemble de tous les symboles terminaux a qui peuvent apparaitre immédiatement à droite de A dans une dérivation.

Méthode

1. Ajouter un marqueur de fin de chaine (symbole \$ par exemple) à SUIVANT(S)
   (où S est l'axiome de départ de la grammaire)
2. Pour chaque production $A \to \alpha B\beta$ où B est un non-terminal, alors
   ajouter le contenu de PREMIER(β) à SUIVANT(B), **sauf** ε
3. Pour chaque production $A \to \alpha B$, alors ajouter SUIVANT(A) à SUIVANT(B)
4. Pour chaque production $A \to \alpha B\beta$ avec $\varepsilon \in$PREMIER(β), ajouter SUIVANT(A) à SUIVANT(B)

Recommencer à partir de l'étape 3 jusqu'à ce qu'on n'ajoute rien de nouveau dans les ensembles SUIVANT.

Algorithme de construction des ensembles SUIVANT

Exemple

Toujours avec la même grammaire **:**

$$\begin{cases} E \to TE' \\ E' \to +TE' \mid -TE' \mid \varepsilon \\ T \to FT' \\ T' \to *FT' \mid /FT' \mid \varepsilon \\ F \to (E) \mid \text{nb} \end{cases}$$

$$\begin{aligned} \text{SUIVANT}(E) &= \{ \$,) \} \\ \text{SUIVANT}(E') &= \{ \$,) \} \\ \text{SUIVANT}(T) &= \{ +, -,), \$ \} \\ \text{SUIVANT}(T') &= \{ +, -,), \$ \} \\ \text{SUIVANT}(F) &= \{ *, /,), +, -, \$ \} \end{aligned}$$

iv) Construction de la table d'analyse LL(1)

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

Méthode

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

Algorithme de construction de la table d'analyse LL (1):

Exemple

Selon la même grammaire précédente, la table d' analyse LL (1) est comme suit:

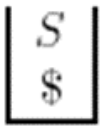
	nb	+	−	*	/	(	)	$
E	$E \to TE'$					$E \to TE'$		
E'		$E' \to +TE'$	$E' \to -TE'$				$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$					$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to \varepsilon$	$T' \to *FT'$	$T' \to /FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to$ nb					$F \to (E)$		

c) Analyseur syntaxique LL(1)

Méthode

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.

- **Données** : mot **m** termine par $, table d'analyse **M**
- **Initialisation de la pile** :



la pile de l 'analyseur syntaxique

```
repeter
        Soit X le symbole en sommet de pile
        Soit a la lettre pointée par ps
        Si X est un non terminal alors
                Si M[X, a] = X → Y1 ... Yn alors
                        enlever X de la pile
                        mettre Yn puis Yn−1 puis ...puis Y1 dans la pile
                        émettre en sortie la production X → Y1 ...Yn
                sinon          (case vide dans la table)
                        ERREUR
                finsi
        Sinon
                Si X =$ alors
                        Si a = $ alors ACCEPTER
                        Sinon ERREUR
                        finsi
                Sinon
                        Si X = a alors
                                enlever X de la pile
                                avancer ps
                        sinon
                                ERREUR
                        finsi
                finsi
        finsi
jusqu'à ERREUR ou ACCEPTER$
```

Algorithme de l'analyseur syntaxique

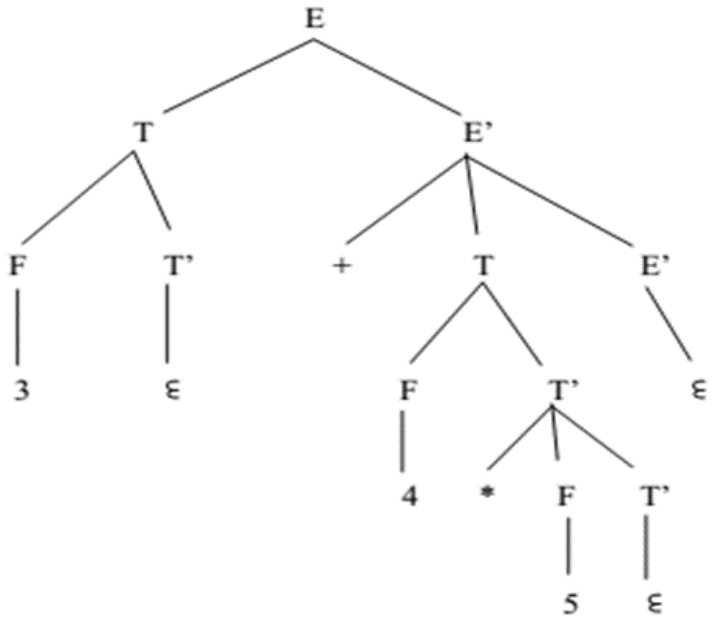
Exemple

Analysez le mot : m= 3 *4 +5, en utilisant la table d'analyse de l'exemple précédent :

PILE	Entrée	Sortie
$ E	$3+4*5$$	$E \to TE'$
$ $E'T$	$3+4*5$$	$T \to FT'$
$ $E'T'F$	$3+4*5$$	$F \to$ nb
$ $E'T'3$	$3+4*5$$	
$ $E'T'$	$+4*5$$	$T' \to \varepsilon$
$ E'	$+4*5$$	$E' \to +TE'$
$ $E'T+$	$+4*5$$	
$ $E'T$	$4*5$$	$T \to FT'$
$ $E'T'F$	$4*5$$	$F \to$ nb
$ $E'T'4$	$4*5$$	
$ $E'T'$	$*5$$	$T' \to *FT'$
$ $E'T'F*$	$*5$$	
$ $E'T'F$	$5$$	$F \to$ nb
$ $E'T'5$	$5$$	
$ $E'T'$	$$$	$T' \to \varepsilon$
$ E'	$$$	$E' \to \varepsilon$
$$	$$$	ACCEPTER (analyse syntaxique réussie)

*La pile pour analyser le mot 3+4*5*

On obtient donc l'arbre syntaxique suivant:



Remarque

- Il est impossible d'appliquer l'algorithme de l'analyse descendante à toutes les grammaires. Effectivement, si la table d'analyse contient plusieurs entrées (**plusieurs productions pour une même case M [A,a]**), il sera impossible de réaliser une telle analyse descendante car il sera impossible de déterminer quelle production appliquer.
- Donc : la grammaire LL (1) est une grammaire où la table d'analyse décrite précédemment ne contient aucune case définie de manière multiple (chaque case contient **au moins une règle de production**).

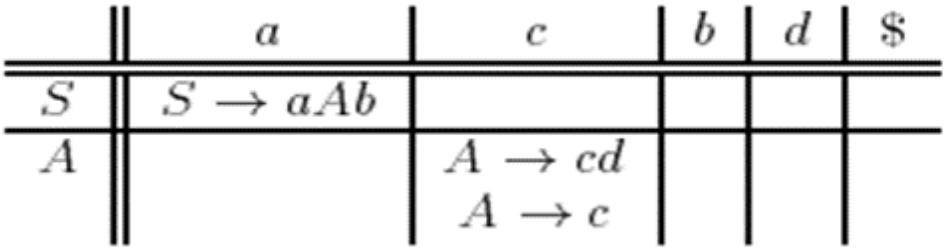
Exemple

Par exemple : on a la grammaire suivant :

S→aAb

A→cd|c

Nous avons PREMIER (S)= {a}, PREMIER(A)={c}, SUIVANT(S)= {$},et SUIVANT(A)={b}, ce qui donne la table d'analyse.



- Il y a deux règles de production pour la case M [A,c], donc ce n'est pas une grammaire LL(1). On ne peut pas appliquer la méthode d'analyse descendante. Donc, pour pouvoir choisir entre la production A→cd et la production A→c, il faut lire la lettre qui suit celle que l'on pointe (donc deux symboles de prévision sont nécessaires).

4.3. Analyse ascendante

a) Introduction

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 il y'a deux opérations à appliquer:

- **Décalage (Shift)** : décaler d'une lettre de pointeur sur le mot en entrée
- **Réduction (reduce)** : réduire une chaîne (suite 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.

Soit la grammaire G ayant les règles de production suivantes :

S→aABe

A→Abc| b

B→d

On se propose d'analyser la chaîne abbcde de manière ascendante :

a b b c d e **décaler**

a b b c d e **réduire**

a A b c d e **décaler**

a A b c d e **décaler**

a A b c d e **réduire**

a A d e **décaler**

a A d e **réduire**

a A B e **décaler**

a A B e **réduire**

S **Analyse réussie**

b) Les analyseurs LR

i) Introduction

L'analyse LR (ou LR(k)) : Il s'agit d'une méthode générale d'analyse syntaxique ascendante, qui repose sur le modèle de décalage-réduction et qui peut être appliquer pour analyser une grande variété de grammaires non contextuelles.

L : **Left to right scanning** (on analyse la chaîne en entrée de la gauche vers la droite)

R : **constructing a Rightmost derivation in reverse** (en construisant une dérivation droite inverse)

k : on utilise k symboles d'entrée de prévision à chaque étape nécessitant la prise d'une décision d'action d'analyse.

Les analyseurs LR comportent :

- Un algorithme d'analyse commun aux différentes méthodes LR.
- Le contenu de la table d'analyse varie en fonction du type d'analyseur LR.

Il y'a trois techniques de construction de tables d'analyse LR pour une grammaire donnée :

Simple LR (SLR), LR canonique, LookAhead LR (LALR).

ii) L'analyse SLR

1 Introduction

- Un analyseur SLR ou LR simple **(left to right, rightmost derivation)** c'est un analyseur pour les grammaires non contextuelles qui lit l'entrée de gauche à droite et produit une dérivation droite.
- La méthode la plus facile à mettre en œuvre consiste à créer une table d'analyse SLR (SLR(1)) à partir d'une grammaire. Elle offre un bon point de départ pour l'analyse LR.
- Selon la méthode SLR, il est essentiel de créer **un automate fini déterministe** à partir de la grammaire, puis de le convertir en une table d'analyse.
- La construction des analyseurs SLR pour une grammaire donnée est basée **sur la collection d'ensembles d'items LR(0)**. Pour construire cette collection, on définit une **grammaire augmentée**, **une fonction fermeture** et une **fonction transition**.

Les étapes à suivre :

- Augmentation de la grammaire S'→S
- On pose l'item I={ S'→.S}
- Calcul de l'item I_0 : I_0 =Fermeture(I)
- Calcul de la transition dans I_0 : *(I_0 ,X)* pour construire des nouveaux items.
- Arrêt jusqu'à ce qu'aucun nouveaux item ne peut plus trouver.
- Construction de la table d'analyse SLR(1)

2 Collection des items LR(0) d'une grammaire

Exemple

Soit la grammaire A des expressions arithmétiques numérotées comme suit:

1. E → E+ T
2. E → T

3. T →T* F
4. T → F
5. F → (E)
6. F → nb

Étape 1 : Augmentation de la grammaire :

S→E

E → E+ T

E → T

T →T* F

T → F

F → (E)

F → nb

Étape 2 : calcul de l'item I

I={S→.E}

Étape 3 : calcul de l'item I_0 :I_0=Fermeture(I)

I_0=Fermeture{S→.E}

Comment calculer la 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 $A \to \alpha.B\beta$
pour chaque production $B \to \gamma$
rajouter (s'il n'y est pas déja) l'item $B \to .\gamma$ dans Fermeture(I)
finpour
finpour
3- Recommencer 2 jusqu'à ce qu'on n'ajoute rien de nouveau

Fermeture d'un ensemble d'items LR(0):

Donc d'après l'algorithme, on obtient :

I_0={S→.E, E → .E+ T, E → .T, T →.T* F, T → .F, F → .(E), F → .nb}

Étape 4 :

- Calcul de la transition dans I_0 : *(I_0,X) :*

$$\Delta(I, X) = \text{Fermeture}(\text{tous les items } A \to \alpha X.\beta) \text{ où } A \to \alpha.X\beta \in I$$

Transition par X d'un ensemble d'items I

On a : I_0={S→.**E**, E → .**E**+ T, E → .**T** , T →.**T*** F, T → .**F**, F → .**(** E), F → .**nb**} Avec : X={E,T,F,(, nb}, donc :

$\Delta(I_0,E)$= {S→E., E →E.+T}=$\mathbf{I_1}$

$\Delta(I_0,T)$= {E → T., T→T.*F}=$\mathbf{I_2}$

$\Delta(I_0,F)$= {T→ F.}=$\mathbf{I_3}$

$\Delta(I_0,()$= {F→(.E), E→.E+T, E→.T, T→.T*F, T→.F, F→.nb, F→.(E)}=$\mathbf{I_4}$

$\Delta(I_0,nb)$={ F →nb.}=$\mathbf{I_5}$

- Calcul de la transition dans I_1 : *(I_1,X) :*

I_1={S→E., E →E.+T}, avec X={+}

Δ(I_1,+)= {E →E+.T, T→.T*F, T→. F, F →.nb, F →.(E) }=**I_6**

- Calcul de la transition dans I_2 : *(I_2,X) :*

I_2={E → T., T→T.*F}, avec X={*}

Δ(I_2,*)={ T→T*.F, F →.nb, F →.(E)}=**I_7**

- Calcul de la transition dans I_3 : *(I_3,X)*

I_3={T→ F.} Ensemble vide de transition

- Calcul de la transition dans I_4 : *(I_4,X)*

I_4={F→(.E), E→.E+T, E→.T, T→.T*F, T→.F, F→.nb, F→.(E)} avec X={E,T,F,(,nb}

Δ(I_4, E)= {F→(E.), E→E.+T}=**I_8**

Δ(I_4, T)={ E→T., T→T.*F }=**I_2**

Δ(I_4, F)={ T→F. }=**I_3**

Δ(I_4,nb)={ F→nb. }=**I_5**

Δ(I_4,()={ F→(.E), E→.E+T, E→.T, T→.T*F, T→.F, F→.nb, F→.(E)}=**I_4**

- Calcul de la transition dans I_5 : *(I_5,X)*

I_5= {F→nb.} Ensemble vide de transition

- Calcul de la transition dans I_6 : *(I_6,X)*

I_6={E →E+.T, T→.T*F, T→. F, F →.nb, F →.(E) } avec X={T,F,(,nb}

Δ(I_6, T)= {E →E+T., T→T.*F }=**I_9**

Δ(I_6, F)={ T→F. }=**I_3**

Δ(I_6, nb)={ F→nb. }=**I_5**

Δ(I_6, ()={ F→(.E), E→.E+T, E→.T, T→.T*F, T→.F, F→.nb, F→.(E)}=**I_4**

- Calcul de la transition dans I_7 : *(I_7,X)* avec X={F,(,nb}

Δ(I_7, F)={ T→T*F.}=**I_{10}**

Δ(I_7, nb)={ F→nb. }=**I_5**

Δ(I_7, ()={ F→(.E), E→.E+T, E→.T, T→.T*F, T→.F, F→.nb, F→.(E)}=**I_4**

- Calcul de la transition dans I_8 : *(I_8,X)* avec X={),+}

Δ(I_8,))= {F→(E).}=**I_{11}**

Δ(I_8, +)= {F→E+.T, T→.T*F, T→. F, F →.nb, F →.(E) }=**I_6**

- Calcul de la transition dans I_9 : *(I_9,X) :*

I_9= {E →E+T., T→T.*F }, avec X={*}

Δ(I_9,*)={ T→T*.F, F →.nb, F →.(E)}=**I_7**

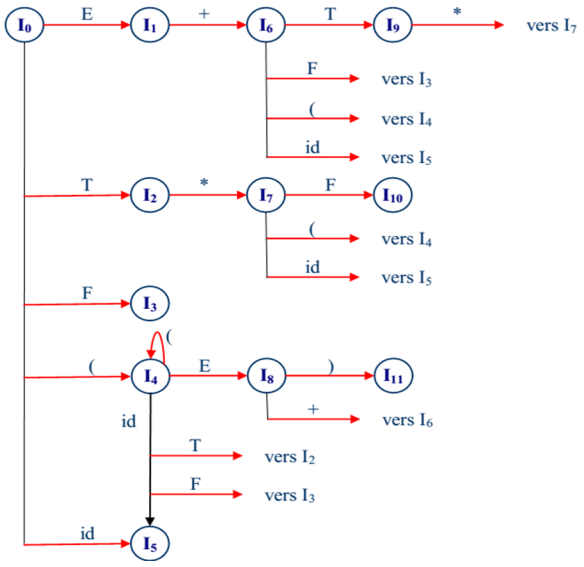
- Calcul de la transition dans I_{10} : *(I_{10},X)*

I_{10}= { T→T*F.}: Ensemble vide de transition

- Calcul de la transition dans I_{11} : *(I_{11},X)*

I_{11}= {F→(E).}: Ensemble vide de transition

- La **fonction de transition** pour l'ensembles d' items de la grammaire augmentée considérée dans cet exemple est donnée sous forme d'un **AFD** représenté comme suit :



La fonction de transition sous forme AFD

3 Construction de la table d'analyse SLR

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*. On note ça **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.

L'algorithme pour construire la table d'analyse SLR :

1- Construire la collection d'items $\{I_o, \ldots, I_n\}$
2- l'état i est contruit à partir de I_i :
a) pour chaque $\Delta(I_i, a) = I_j$: mettre `decaller j` dans la case $M[i, a]$
b) pour chaque $\Delta(I_i, A) = I_j$: mettre `aller en j` dans la case $M[i, A]$
c) pour chaque $A \to \alpha.$ (sauf $A = S'$) contenu dans I_i :
mettre `reduire` $A \to \alpha$ dans **chaque** case $M[i, a]$ où $a \in$SUIVANT(A)
d) si $S' \to S. \in I_i$: mettre `accepter` dans la case $M[i, \$]$

L'algorithme pour construire la table d'analyse SLR

Toujours le même exemple : l'ensemble des SUIVANT et PREMIER :

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

Et donc la table d'analyse **SLR** de cette grammaire est :

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			

Table d'analyse SLR

4 Analyseur syntaxique SLR

On commence par l'état 0 et on empile et dépile non seulement les symboles (comme dans l'analyseur LL), mais également les états successifs.

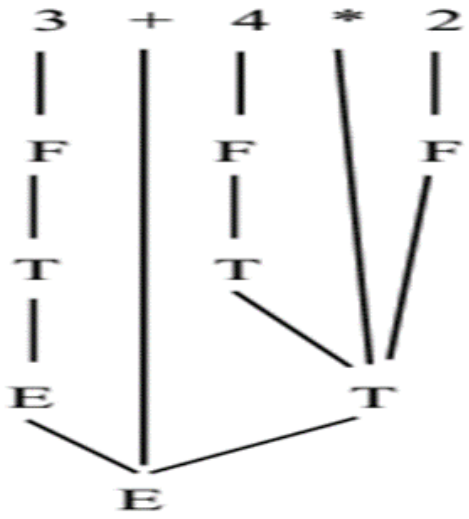
Exemple

L'analyse du mot 3+4*2$ est comme suit :

pile	entrée	action
$ 0	$3+4*2$	d5
$ 0 3 5	$+4*2$	r6 : $F \to$ nb
$ 0 F	$+4*2$	je suis en 0 avec F : je vais en 3
$ 0 F 3	$+4*2$	r4 : $T \to F$
$ 0 T	$+4*2$	je suis en 0 avec T : je vais en 2
$ 0 T 2	$+4*2$	r2 : $E \to T$
$ 0 E	$+4*2$	je suis en 0 avec E : je vais en 1
$ 0 E 1	$+4*2$	d6
$ 0 E 1 + 6	$4*2$	d5
$ 0 E 1 + 6 4 5	$*2$	r6 : $F \to$ nb
$ 0 E 1 + 6 F	$*2$	je suis en 6 avec F : je vais en 3
$ 0 E 1 + 6 F 3	$*2$	r4 : $T \to F$
$ 0 E 1 + 6 T	$*2$	en 6 avec T : je vais en 9
$ 0 E 1 + 6 T 9	$*2$	d7
$ 0 E 1 + 6 T 9 * 7	2	d5
$ 0 E 1 + 6 T 9 * 7 2 5	$	r6 : $F \to$ nb
$ 0 E 1 + 6 T 9 * 7 F	$	en 7 avec F : je vais en 10
$ 0 E 1 + 6 T 9 * 7 F 10	$	r3 : $T \to T * F$
$ 0 E 1 + 6 T	$	en 6 avec T : je vais en 9
$ 0 E 1 + 6 T 9	$	r1 : $E \to E + T$
$ 0 E	$	en 0 avec E : je vais en 1
$ 0 E 1	$	ACCEPTÉ !!!

Pile de l'analyseur syntaxique SLR

Donc l'arbre de dérivation du ce mot 3+4*2 est comme suit :



Arbre de dérivation ascendante

5 Analyse LR(0)

Méthode

Les étapes à suivre :

- Collection **des items LR(0)** d'une grammaire comme l'indiqué dans la partie de l'analyse SLR.
- Construction de la **table d'analyse LR(0)** comme la table d'analyse SLR, mais la seul différence est dans le cas : Si un groupe d'items i comprend un item de la forme A → w • où A → w est la règle numéro m, alors la ligne de l'état i dans la **partie des symboles terminaux est entièrement remplie par la réduction rm**. C'est pourquoi ce sont les tables d'analyse syntaxique LR(0) : elles n'anticipent pas (c'est-à-dire qu'elles ne tiennent compte d'aucun symbole d'avance) pour déterminer quelle réduction faire.

Exemple

Pour la même grammaire de l'exemple du partie de l'analyse SLR:

1- On a 11 items LR (0) (I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9, I_{10} , I_{11})

2- La table d'analyse LR(0) est comme suit:

	nb	+	*	(	)	$	E	T	F
0	D5			D4			1	2	3
1		D6				acc			
2	R2	R2	R2/d7	R2	R2	R2			
3	R4	R4	R4	R4	R4	R4			
4	D5			D4			8	2	3
5	R6	R6	R6	R6	R6	R6			
6	D5			D4				9	3
7	D5			D4					10
8		D6			D11				
9	R1	R1	R1/d7	R1	R1	R1			
10	R3	R3	R3	R3	R3	R3			
11	R5	R5	R5	R5	R5	R5			

iii) L'analyse LR(1)

1 Introduction

L'analyse LR canonique ou LR(1) permet d'associer plus d'informations à un état afin d'éviter quelques actions invalides et donc des conflits.s

Les étapes à suivre :

- Augmentation de la grammaire (grammaire augmenté G') S'→S
- On pose l'item I={ S'→.S, $}

- Calcul de l'item I_0 : I_0 =Fermeture(I)
- ·Calcul de la transition dans I_0 : *(I_0 ,X)* pour construire des nouveaux items.
- Arrêt jusqu'à ce qu'aucun nouveaux item ne peut plus trouver.
- Construction de la table d'analyse LR(1).

2 Collection des items LR(1) d'une grammaire

Exemple

Soit la grammaire G des expressions arithmétiques numérotées comme suit:

1. S → CC
2. C → cC|d

Étape 1 : Augmentation de la grammaire :

S'→S

1. S → CC
2. C → cC
3. C → d

Étape 2 : calcul de l'item I

I={S'→ .S, $}

Étape 3 : calcul de l'item I_0 :I_0=Fermeture(I)

I_0=Fermeture {S'→ .S, $}

Comment calculer la **Fermeture** d'un ensemble d'items I ?

```
Fonction Ferm(I);
Début
  Répéter
    Pour chaque item [A→α.Bβ, a] ∈ I, chaque production B → δ ∈ G'
    et chaque terminal b∈Prem(βa) tel que [B→.δ, b] ∉ I
    Faire Ajouter [B → .δ, b] à I
  Jusqu'à ce qu'aucun item ne puisse être ajouté à I;
  Ferm(I) := I;
Fin ;
```

Fermeture d'un ensemble d'items LR(1)

Donc d'après l'algorithme, on obtient :

I_0={S'→.S, $

S → .CC, $

C → .cC, c|d

C → .d, c|d }

Étape 4:

- Calcul de la transition dans I_0 : *(I_0,X) :*

```
Fonction Trans(I, X);
Début
  Soit J l'ensemble des items [A→αX.β, a] tels que [A→α.Xβ, a] ∈ I.
  Trans(I, X) := Ferm(J);
Fin ;
```

Transition par X d'un ensemble d'items I

$\Delta(I_0,S)$= {S'→S., \$}=**I_1**

$\Delta(I_0,C)$= {S → C.C, \$

C→.cC, \$

C→.d, \$} =**I_2**

$\Delta(I_0,c)$= {C→ c.C, c|d

C→.cC, c|d

C→.d, c|d } =**I_3**

$\Delta(I_0,d)$= {C→d., c|d } =**I_4**

- Calcul de la transition dans I_2 : *(I_2,X) :*

$\Delta(I_2,C)$= {S→ CC., \$} =**I_5**

$\Delta(I_2,c)$= {C→ c.C, \$

C→.cC, \$

C→.d, \$ } =**I_6**

$\Delta(I_2,d)$= {C→ d., \$} =**I_7**

- Calcul de la transition dans I_3 : *(I_3,X) :*

$\Delta(I_3,C)$= {C→ cC., c|d}=**I_8**

$\Delta(I_3, c)$= {C→ c.C, c|d

C→.cC, c|d

C→.d, c|d } =**I_3**

$\Delta(I_3, d)$= {C→ d., c|d}=**I_4**

- Calcul de la transition dans I_6 : *(I_6,X) :*

$\Delta(I_6, C)$= {C→ cC., \$}=**I_9**

$\Delta(I_6,c)$= {C→ c.C, \$

C→.cC, \$

C→.d, \$ } =**I_6**

$\Delta(I_6,d)$= {C→ d., \$} =**I_7**

3 Table d'analyse LR(1)

Méthode

L'algorithme pour construire la table d'analyse LR(1) :

Algorithme ConstructionTableLR;
Début
1/ L'état i est construit à partir de I_i. La partie ACTION pour l'état i est déterminée comme suit:
a. **Si** $[A \rightarrow \alpha.a\beta, b] \in I_i$ et Trans $(I_i, a) = I_j$ (avec $a \in V_T$) **Alors** ACTION [i, a] $\leftarrow d_j$ (décaler j)
b. **Si** $[A \rightarrow \alpha., a] \in I_i$ (avec $A \neq S'$) **Alors** ACTION [i, a] $\leftarrow r_{num}$
(réduire par la régle $A \rightarrow \alpha$ dont le numéro est num)
c. **Si** $[S' \rightarrow S., \$] \in I_i$ **Alors** ACTION [i, $] $\leftarrow$ accepter
2/ La partie SUCCESSEUR pour l'état i est déterminée comme suit :
Si Trans $(I_i, A) = I_j$ **Alors** SUCCESSEUR [i, A] $\leftarrow$j
3/ Toutes les entrée restantes dans la table sont mises à ERREUR
4/ L'état initial de l'analyseur est construit à partir de l'ensemble d'items contenant $[S' \rightarrow .S, \$]$
Fin ;

L'algorithme pour construire la table d'analyse LR(1)

Et donc la table d'analyse LR(1) de cette grammaire est :

	c	d	$	S	C
0	D3	D4		1	2
1			accept		
2	D6	D7			5
3	D3	D4			8
4	R3	R3			
5			R1		
6	D6	D7			9
7			R3		
8	R2	R2			
9			R2		

iv) L'analyse LALR

1 Introduction

- Cet algorithme se base sur la construction de la collection d'ensembles d'items LR(1) ainsi que la table d'analyse LR(1). Il utilise la notion de cœur d'item LR(1), sachant que : **item LR(1)=[cœur, symbole de pré-vision].**
- L'objectif principal de cet algorithme est de **créer les ensembles d'éléments LR(1)** et de fusionner les ensembles qui possèdent un **cœur commun.** La table d'analyse LALR(1) est créée en associant les ensembles d'items fusionnés.

2 Algorithme de construction table d'analyse LALR

Méthode

Algorithme ConstructionTableLALR;
Début
1/ Construire la collection d'ensembles d'items LR(1) pour la grammaire augmentée G'.
2/ Pour chaque cœur présent parmi les ensembles d'items LR(1), trouver tous les états ayant ce même cœur et remplacer ces états par leur union.
3/ La table LALR(1) est obtenue en condensant la table LR(1) par superposition des lignes correspondant aux états regroupés.
Fin ;

Algorithme de construction table d'analyse LALR

Exemple

On prend la grammaire G traitée dans la section 1.3 précédente pour la construction de la table d'analyse LALR(1) :

S →CC

C→cC | d

- Pour l'ensembles d'items LALR(1), on recherche les ensembles d'items LR(1) qui peuvent être fusionnés :

1- $\mathbf{I_3+I_6}$={C→ c.C, c|d|$

C→.cC, c|d|$

C→.d, c|d|$ }

2- $\mathbf{I_4+I_7}$= {C→ d., c|d|$}

3- $\mathbf{I_8+I_9}$= {C→ cC., c|d|$}

- La table d'analyse LALR(1)

	c	d	$	S	C
0	D36	D47		1	2
1			accept		
2	D36	D47			5
36	D36	D47			89
47	R3	R3	R3		
5			R1		
89	R2	R2	R2		

4.4. Utilisation des priorités et des associativités pour résoudre les actions conflictuelles d'analyse

Le grammaire ambigüe fait des conflits dans la table d'analyse:

- Conflit décalage /réduction : Il est impossible de déterminer à la lecture du terminal « a » s'il est nécessaire de réduire la production S→σ ou de déplacer le terminal.
- Conflit réduction /réduction : Il est impossible de déterminer à la lecture du terminal 'a' s'il est nécessaire de réduire une production S→σ ou une production T→B.

→ Il est donc nécessaire de résoudre les conflits en accordant des **priorités** aux actions. (décaler ou réduire) et aux productions.

Par exemple, soit la grammaire : E→E+E|E*E|nb

1) Soit à analyser 3+4+5. Lorsqu'on lit le 2iéme + on a le choix entre :

- réduire ce qu'on a déjà lu par : E→E+E. Ce qui nous donnera finalement le calcul (3+4) +5
- décaler ce +, ce qui nous donnera finalement le calcul : 3+ (4+5).

→ + est associatif à gauche, donc on préférera réduire.

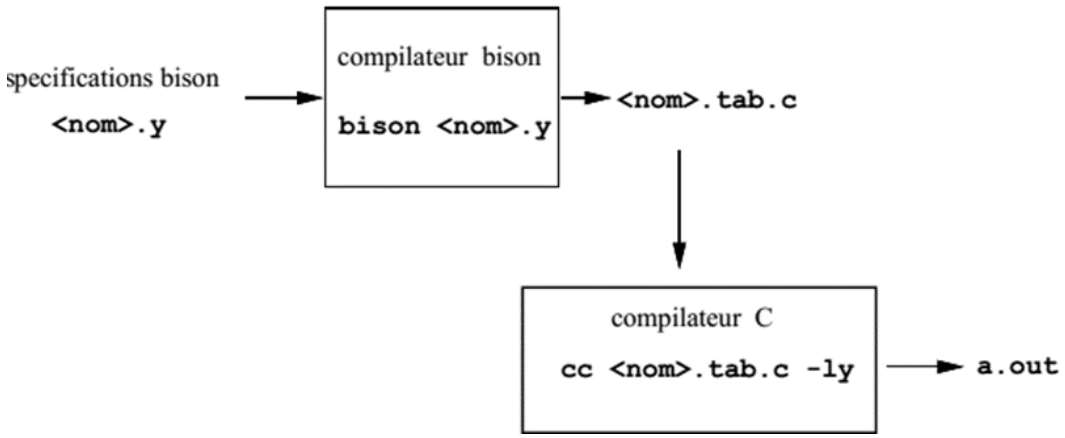
2) Soit à analyser 3+4*5. Lorsqu'on lit le * on a encore un choix shift/reduce. Si l'on réduit on calcule (3+4)*5, si on décale on calcule 3+ (4*5) : donc Il faut décaler → car le * est prioritaire que +.

3) Soit à analyser, 3*4+5 : donc il faut réduire→ il faut mettre quelque part dans l'analyseur le fait que * est prioritaire sur +.

4.5. Générateur d'analyseur syntaxique : YACC

a) Introduction

- Plusieurs méthodes ont été développées afin de créer des analyseurs syntaxiques à partir de grammaires. En d'autres termes, des outils qui créent automatiquement une table d'analyse en utilisant une grammaire spécifique.
- Yacc/bison accepte la description d'un langage sous forme de règles de production et crée un programme écrit dans un langage de haut niveau (le langage C). Après sa compilation, il reconnaît des phrases de ce langage (ce programme est un analyseur syntaxique).



Fonctionnement de YACC

b) Le format d'une spécification YACC

Format général

%{

Déclarations : 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

%%

Bloc principal

- **Partie Déclaration**

1. La partie déclarations contient les déclarations "C" (entre %{ et %}) ainsi que la déclaration des noms de tokens : **%token** name1 name2
2. Dans la partie déclarative, tout nom non défini est considéré comme un symbole non terminal. Il est nécessaire que tout symbole non-terminal soit présent au moins une fois dans la partie gauche d'une règle.
3. À la place de tokens, on utilise **left** et **right** lorsque les tokens correspondent à des opérateurs pour lesquels on veut définir une propriété **d'associativité**.

Exemple

on écrira :

%left '+' '-' /* addition et soustraction associatives à gauche */

%right '^' /* exponentiation associative à droite */

Les symboles ont des priorités selon l'ordre dans lequel ils sont présentés dans leur déclaration d'associativité, les premiers ayant la plus faible. Quand les symboles apparaissent dans la même déclaration d'association, ils sont prioritaires.

- **Les règles ont le format suivant :**

A : CORPS;

"A" représente un symbole non-terminal, et **"CORPS"** représente une chaîne de noms et lettres. Les symboles **":"** et **";"** sont des délimiteurs. Les noms peuvent être des symboles terminaux ou non-terminaux. YACC nécessite que les noms des symboles terminaux soient déclarés ainsi dans la partie de déclarations.

Si plusieurs règles ont la même partie gauche, on peut utiliser le caractère " | ".

- **Actions**

Chaque règle syntaxique est liée à des actions. Ces actions peuvent générer diverses valeurs et peuvent exploiter les valeurs générées par d'autres actions. Une action est un morceau de code en C, placé entre des accolades.

G : S B 'X' { printf("mot reconnu");}

- **Le programme en C**

Il doit inclure le programme principal main () qui doit généralement appeler la fonction yyparse(). Yacc a développé cette fonction qui doit inclure une fonction yylex qui réalise l'analyse lexicale du texte. cette fonction est appelé par l'analyseur syntaxique à chaque fois qu'il a besoin du terminal suivant.

c) 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***.
- Lors d'une action lexicale (comme dans le fichier lex), l'instruction return(ul) permet de renvoyer l'unité lexicale ul à l'analyseur syntaxique. Cette unité lexicale peut être classée dans le **yylval**.
- Le contenu de yylval sera automatiquement considéré par l'analyseur syntaxique comme étant la valeur de l'attribut associé à cette unité lexicale. yylval est une variable de type YYSTYPE (indiquée dans la bibliothèque yacc/bison) qui est initialement une int. On peut changer ce type par: **#define YYSTYPE autre_type_C** ou encore par : **%union {champs d'une union C}**

Par exemple

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

Ce exemple permet de stocker dans yylval à la fois des int, des double et des char*.

Pour faciliter la communication entre actions et l'analyseur syntaxique on utilise **le symbole $** de la manière suivante :

- A chaque symbole (terminal ou non) est associe une valeur (de type entier par défaut). Cette valeur peut être utilisée dans les actions sémantiques.
- Le symbole **$$** référence la valeur de l'attribut associe au non terminal de la partie gauche.
- **$$_i$$** référence la valeur associée au i-éme symbole (terminal ou non terminal), ou action sémantique de la partie droite.

d) Variables, fonctions et actions prédéfinies

YYPARSE() : appel de l'analyseur syntaxique. yyparse retourne alors 1, ce qui peut être utilisé pour indiqué l'échec de l'analyseur.

YYACCEPT : permettant de stopper l'analyseur syntaxique.

main() : le main par défaut se contente d'appeler yyparse(). L'utilisateur peut écrire sa propre main dans la partie du bloc principal.

yyerror() fonction définie dans la bibliothèque de YACC, qui indique qu'une erreur de syntaxe a été rencontrée.

Exemple

- **La source Lex du mini-interprète d'expressions :**

```
%{
#include "global.h"
#include "calc.h"
#include <stdlib.h>
%}
chiffre   [0-9]
```

```
entier   {chiffre}+
exposant  [eE][+-]?{entier}
reel   {entier}("."{entier})?{exposant}?
%%
{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);
```

- **Le source YACC du mini interprète d'expressions :**

```
%{
#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(); }
```

5. Exercices

5.1. Exercice 01

Soit la grammaire d'expressions arithmétiques **A** suivante:

où T :{ ; + - * / const ( ) }, N={S,E,T,F}

1. Donner les dérivations les plus à gauche pour la chaîne 5+3*2 ;

$$
\begin{array}{lcl}
S & \longmapsto & E \;;\; S \mid \epsilon \\
E & \longmapsto & E + T \mid E - T \mid T \\
T & \longmapsto & T * F \mid T / F \mid F \\
F & \longmapsto & \texttt{const} \mid (\; E \;)
\end{array}
$$

Grammaire A

5.2. Exercice 02

1. Soit G (T, N, A, P), tel que : T :{a, b}, N={A, B, C} et A : est l'axiome de départ.

- Supprimer la récursivité à gauche dans la grammaire G.

$$
\begin{array}{lcl}
A & \rightarrow & BC \mid a \\
B & \rightarrow & CA \mid Ab \\
C & \rightarrow & AB \mid CC \mid a
\end{array}
$$

Grammaire G

2. Soit les deux grammaires suivantes :

S →*bA|*bC|*F où T :{*,b}, N={A, C,F} (grammaire 1)

$$G = \langle \{E, S\}, \{i, t, e, a, b\}, \left\{ \begin{array}{c} S \to iEtS \mid iEtSeS \mid a, \\ E \to b \end{array} \right\} \rangle$$

Grammaire 2

- Appliquer la factorisation à gauche pour ces deux grammaires.

5.3. Exercice 03

Soit la grammaire :

$$\begin{array}{lcl} S & \longmapsto & AaB \\ A & \longmapsto & CB \mid Bb \mid \epsilon \\ B & \longmapsto & b \\ C & \longmapsto & c \mid \epsilon \end{array}$$

1. Donner les ensembles PREMIER et SUIVANT pour cette grammaire.
2. Construire la table d'analyse et dire a ce que cette grammaire est LL(1) ?

5.4. Exercice 04

Dites en justifiant votre réponse, si les grammaires suivantes sont LL(1). Si la réponse est oui, analyser le mot ibta pour la grammaire 1 et le mot (c+c) pour la grammaire 2.

1.
$$\begin{array}{l} S \to iEtSS' \mid a \\ S' \to eS \mid \epsilon \\ E \to b \end{array}$$

2.
$$\begin{array}{l} E \to (EOE) \mid c \\ O \to + \mid * \end{array}$$

5.5. Exercice 05

Soit la grammaire d'expressions arithmétiques suivante :

$$\begin{array}{lcl} S & \longmapsto & E \ ; \ S \mid \epsilon \\ E & \longmapsto & T \ E2 \\ E2 & \longmapsto & + \ T \ E2 \mid - \ T \ E2 \mid \epsilon \\ T & \longmapsto & F \ T2 \\ T2 & \longmapsto & * \ F \ T2 \mid / \ F \ T2 \mid \epsilon \\ F & \longmapsto & \texttt{const} \mid \texttt{(} \ E \ \texttt{)} \end{array}$$

1. Calculer l'ensemble des premiers et suivants
2. Simuler l'analyse de l'expression **3*(1+2)/(1) ;** par un analyseur de type descendant.

5.6. Exercice 06

Soit la grammaire G suivante :

S→G=D | D

G →*D | id

D →G

1. Montrer si cette grammaire est SLR(1) ? Si oui, analyser les mots *id==id* et *id=id*

5.7. Exercice 07

Soit la grammaire G, ayant les règles de production suivantes :

L→E|E,L

E→ Є|x

1. Montrer si cette grammaire est LR(0) ou SLR(1)?

5.8. Exercice 08

Soit la grammaire :

$$
\begin{array}{lcl}
< inst > & \longmapsto & IF < expression > THEN < inst >< else - inst > \\
< inst > & \longmapsto & ID := ID \\
< else - inst > & \longmapsto & ELSE < inst > \\
< else - inst > & \longmapsto & \epsilon \\
< expression > & \longmapsto & ID
\end{array}
$$

Les terminaux sont : **IF**, **THEN**, **ID**, **:=**, **ELSE**

1. A ce que cette grammaire est LR(0) ?
2. A ce que cette grammaire est SLR(1) ?

5.9. Exercice 09

Soit la grammaire G, ayant les règles de production suivantes :

S→T|xb

T→aTb|x

1) Montrer si cette grammaire est LR(1)?

2) Analyser les mots suivantes :aaxbb, aaxb

5.10. Exercice 10

Soit la grammaire G, ayant les règles de production suivantes :

S→S+S

S→id

1. Montrer si cette grammaire est LR(1)?

Traduction dirigée par la syntaxe

1. Introduction

L'analyseur syntaxique s'assure que les instructions sont bien formées.

Mais l'analyseur syntaxique ne peut pas vérifier certaines propriétés fondamentales d'un langage. De plus, il est impossible de décrire ces caractéristiques du langage source en utilisant uniquement la grammaire hors contexte du langage.

Par exemple :

- Il est impossible d'utiliser une variable sans qu'elle soit préalablement déclarée.
- Il est impossible de déclarer une variable à plusieurs fois dans le même bloc.
- Il est essentiel de respecter le nombre et les types de paramètres lors d'un appel de fonction.
- Il est impossible de réaliser certaines opérations sur certains types. Par exemple : Il est impossible de multiplier un réel avec une chaîne.

L'analyseur sémantique permet donc de vérifier ces contraintes. En règle générale, elle est réalisée simultanément à l'analyse syntaxique, en utilisant des actions sémantiques insérées dans les règles de production, c'est-à-dire en utilisant des définitions dirigées par la syntaxe (DDS).

2. Grammaire Attribuée

- Une grammaire attribuée est une grammaire hors contexte qui comprend des attributs, des règles sémantiques et des conditions.
- Il s'agit donc d'un formalisme qui permet d'ajouter des actions (règles sémantiques) aux règles de production d'une grammaire. il s'agit de définition dirigée par la syntaxe

3. Définition dirigée par la syntaxe

- Une définition dirigée par la syntaxe (**DDS**) est un formalisme permettant d'ajouter des actions à une règle de production de la grammaire.
- On associe un ensemble d'attributs (valeurs, types) pour chaque symbole de la grammaire.
- 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.
- Chaque règle de production de la grammaire comporte un ensemble d'actions permettant de déterminer la valeur des attributs. La grammaire et l'ensemble des règles sémantiques constituent **la définition dirigée par la syntaxe**.
- Une règle sémantique est un ensemble d'instructions écrites dans un langage spécifique (C, java …). elle peut inclure des affectations, des instructions conditionnelles (si, sinon), des instructions d'affichage.

Exemple

Exemple de règles sémantiques

En supposant que :

- Toutes les symboles non terminales ont un attribut **valeur** qui fournit la valeur calculée ou héritée de l'expression correspondante.
- Chaque terminal dispose d'un attribut **valeur_lexicale** qui donne la valeur du symbole telle qu'elle est lue dans le fichier du code source du programme à analyser (valeur du lexème).

production	action sémantique
$E \rightarrow E + T$	E .val:=E .val+T.val
$E \rightarrow E - T$	E .val:=E .val−T.val
$E \rightarrow T$	E.val:=T.val
$T \rightarrow T * F$	T .val:=T .val∗F.val
$T \rightarrow F$	T.val:=F.val
$F \rightarrow (E)$	F.val:=E.val
$F \rightarrow$ nb	F.val:=nb

Exemple d'un définition dirigé par la syntaxe

4. Représentation par arbres

4.1. Arbre syntaxique décoré

Un arbre syntaxique décoré est un arbre syntaxique où la valeur de chaque attribut est ajoutée aux nœuds.

Exemple

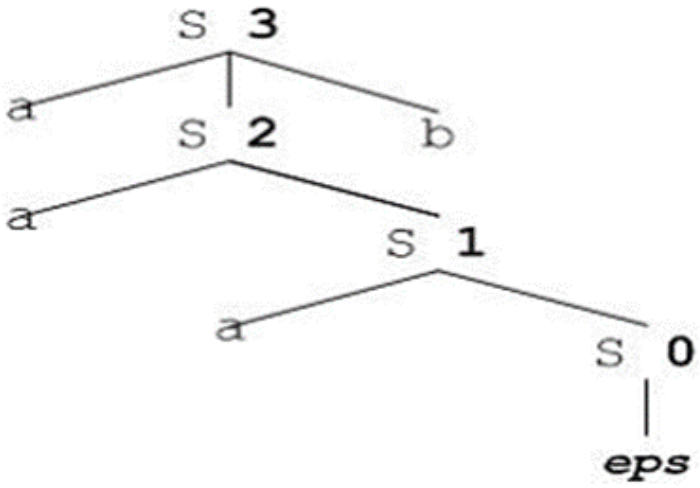
Soit la grammaire : S→aSb|aS| ε attribut : nba(calcul du nombre de a) :

Une DDS permettant de calculer ces attributs pourrait être :

Production	Règle sémantique
$S \longrightarrow aSb$	$S.nba := S_1.nba + 1$
$S \longrightarrow aS$	$S.nba := S_1.nba + 1$
$S \longrightarrow \epsilon$	$S.nba := 0$

DDS permettant de calculer l'attribut a

- Dans cet exemple de DDS, les attributs des symboles en parties gauches dépendent des attributs des symboles de la partie droite.
- On peut dessiner un arbre syntaxique avec la valeur de l'attribut nba pour chaque symbole non terminal. Par exemple pour le mot aaab



Arbre syntaxique avec la valeur de l'attribut nba

On peut observer sur cet arbre syntaxique que **les attributs d'un nœud** peuvent être calculés une fois que **les attributs de tous ses fils** ont été calculés.

4.2. Arbre syntaxique abstrait

- L'analyse syntaxique consiste à créer un arbre de dérivation syntaxique afin de représenter les phrases bien structurées d'un langage. On désigne cet arbre sous le nom d'arbre syntaxique concret.
- Les non-terminaux de l'arbre syntaxique **concret** constitue dans les règles de la grammaire ne sont pas considérés comme utiles pour l'analyse sémantique. En les supprimant, on obtient donc de nouvelles représentations d'un programme (arbre syntaxique **abstrait**).
- L'arbre syntaxique **abstrait** est élaboré à partir de l'arbre syntaxique de dérivation et ne comprend aucun symbole non terminal. Il est élaboré en utilisant les actions sémantiques des règles de production de l'analyseur syntaxique.

Exemple

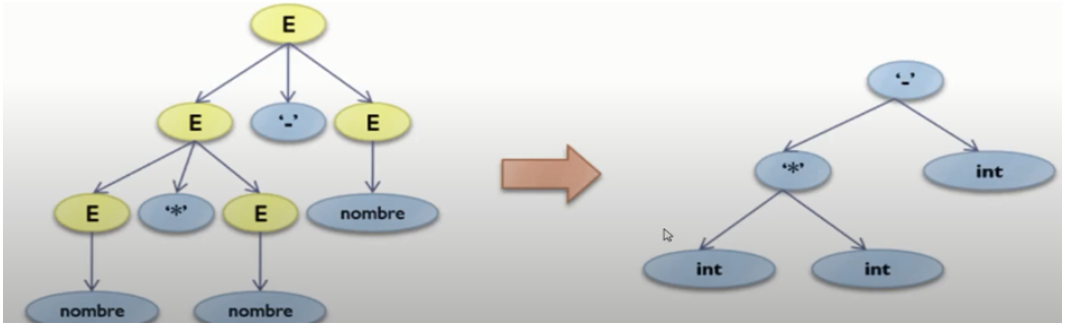
Soit la grammaire suivant pour l'expression 3*4-7 :

E→E-E

|E*E

|nombre

- L'arbre dans la partie gauche c'est un arbre syntaxique concret
- L'arbre dans la partie droite c'est un arbre syntaxique abstrait



Arbre syntaxique abstraite et concret

Construction d'un arbre abstrait

- Créer un nœud dont l'étiquette est « **opérateur** » et avec deux champs de pointeurs vers les nœuds des opérandes gauche et droite
- Créer un nœud dont l'étiquette est « **identifiant** » et avec un champ pour un pointeur vers une entrée de la table des symboles (afin de pouvoir récupérer la valeur)
- Créer un nœud dont l'étiquette est « **nombre** » et avec un champ pour la valeur de la constante

Exemple

Construire l'arbre pour l'expression a-4+c

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)

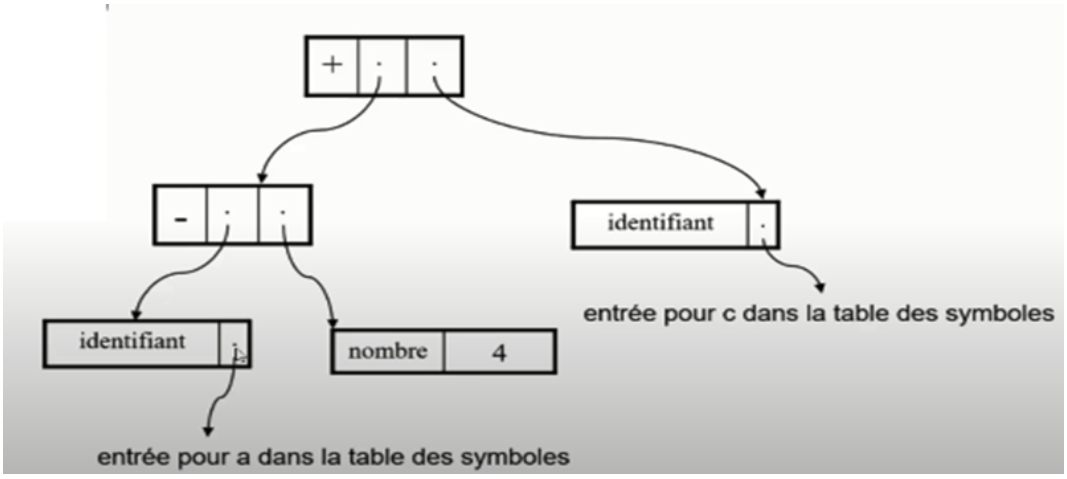
Pointeur1→créerFeuille(identifiant,ptr(a))

Pointeur2→créerFeuille(nombre,4)

Pointeur3→créerNoeud('-',Pointeur1, Pointeur2)

Pointeur4→créerFeuille(identifiant,ptr(c))

Pointeur5→créerNoeud('+',Pointeur3, Pointeur4)



Construction d'un arbre syntaxique abstrait pour a-4+c

4.3. Types d'attributs

Attribut synthétisé

Définition

Le symbole en partie gauche est rattaché à un attribut synthétisé qui est calculé à partir des attributs des symboles de la partie droite (de ses fils).

Attribut hérité *Définition*

Un attribut hérité est obtenu en calculant les attributs du non terminal de la partie gauche et les attributs des autres symboles de la partie droite (des attributs du nœud père et des attributs des nœuds frères).

DDS S-attribuée *Définition*

La définition dirigée par la syntaxe qui ne contient que des attributs synthétisés est connue sous le nom de définition **S-attribuée.**

Évaluation des attributs dans DDS (S-attribuée) : *Méthode*

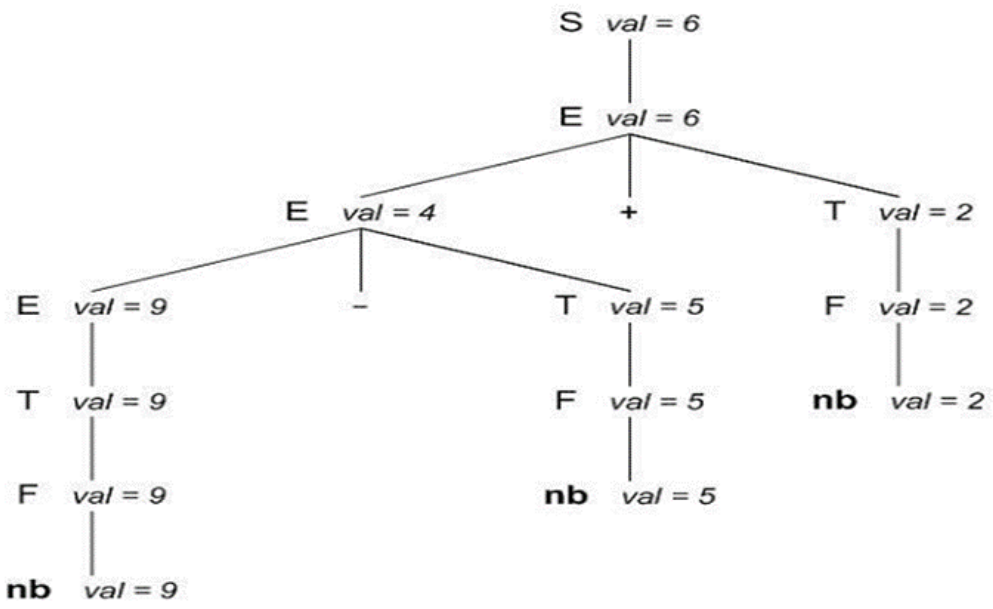
La traduction de définitions **S-attribuées** est parfaitement adaptée à l'analyse ascendante, car l'arbre syntaxique est construit en **feuilles** vers la **racine**.

Exemple

Soit la DDS (S-attribuée) suivante pour analyser le mot : 9 - 5 + 2

Production	Règle sémantique
$S \longrightarrow E\$$	S.val := E.val
$E \longrightarrow E + T$	E.val := E_1.val+T.val
$E \longrightarrow E - T$	E.val := E_1.val-T.val
$E \longrightarrow T$	E.val := T.val
$T \longrightarrow T * F$	T.val := T_1.val*F.val
$T \longrightarrow F$	T.val := F.val
$F \longrightarrow (E)$	F.val := E.val
$F \longrightarrow nb$	F.val := nb.val

La figure suivante montre la construction de l'arbre de dérivation à partir des feuilles (analyse ascendante):



Arbre décoré ascendant

DDS L-attribuée *Définition*

Une définition dirigée par la syntaxe est L-attribuée si tout attribut hérité d'un symbole de la partie droite d'une production ne dépend que :

- des attributs hérités du symbole en partie gauche
- des attributs des symboles le précédant dans la production.

Évaluation des attributs dans DDS (L-attribuée) *Définition*

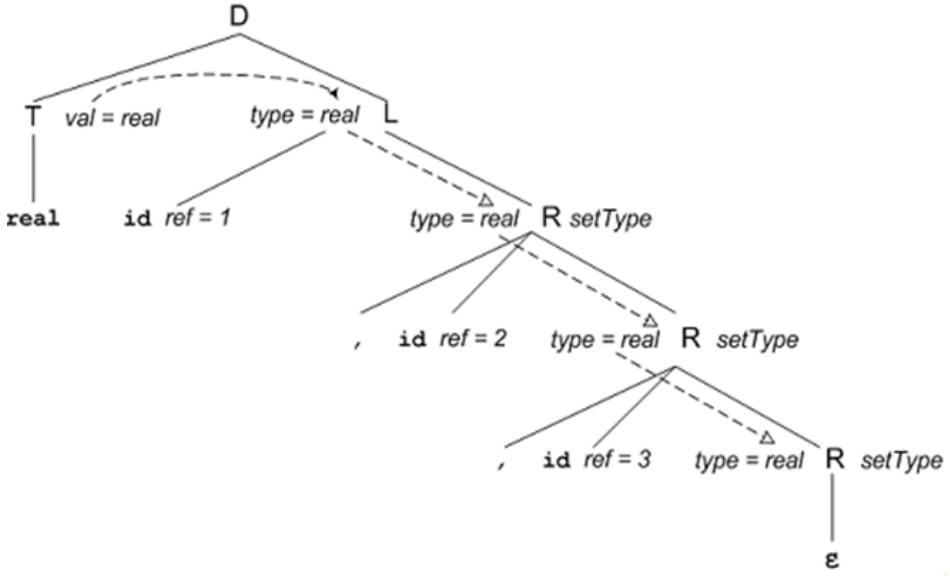
La méthode d'analyse **descendante** est adaptée à la traduction de définitions **L-attribuées** qui ne possèdent que des attributs hérités, car l'arbre syntaxique est construit de la **racine** vers les **feuilles**.

Exemple

Soit la DDS (L-attribuée) suivante pour la déclaration des variables en C pour la déclaration: real x,y,z

Production	Règle sémantique
$D \longrightarrow T\ L$	L.type = T.val
$L \longrightarrow id\ R$	$R.type = L.type$ setType(id.ref,L.type)
$R \longrightarrow, id\ R$	$R_1.type = R.type$ setType(id.ref,R.type)
$R \longrightarrow \epsilon$	
$T \longrightarrow$ int	T.val = integer
$T \longrightarrow$ real	T.val = real

La figure suivante montre la construction de l'arbre de dérivation à partir de la racine (analyse descendante) :



Arbre décoré descendant

5. Exercices

5.1. Exercice 01

Soit la grammaire suivante d'expressions arithmétiques définie par les productions numérotées comme suit :

$$
\begin{aligned}
E &\longmapsto E + T \\
E &\longmapsto T \\
T &\longmapsto T * F \\
T &\longmapsto F \\
F &\longmapsto (E) \\
F &\longmapsto \mathrm{nb}
\end{aligned}
$$

1. Programmez cette grammaire avec lex & yacc
2. Ajouter des actions sémantiques à la grammaire yacc pour calculer la valeur d'une expression arithmétique écrite avec la grammaire précédente.

5.2. Exercice 02

Soit la DDS suivante :

Production	Règles sémantiques
T→FT'	T'.inh=F.val T.val=T'.syn
T'→*FT'$_1$	T'$_1$.inh=T'.inh*F.val T'.syn=T'$_1$.syn
T'→ ε	T'.syn=T'.inh
F→digit	F.val=digit.val

1. Donnez l'arbre de dérivation décoré pour la chaine **3*5**

5.3. Exercice 03

Soit la grammaire suivante :

S→aSb|aS|cSacS| ε Et les attributs : nba(nombre de a) nbc (nombre de c).

production	Règle sémantique
S→aSb	?
S→aS	?
S→cSacS	?
S→ ε	?

1. Etablir une DDS permettant de calculer ces attributs (nombre de a et de c)
2. Donnez l'arbre de dérivation décoré pour la chaine d'entrée **acaacabb**

5.4. Exercices 04

Considérons la définition dirigée par la syntaxe (DDS) d'une imbrication de parenthèses suivante :

Production	Règle sémantique
$S' \longrightarrow S\$$	S.nb := 0
$S \longrightarrow (S)S$	S_1.nb := S.nb + 1 S_2.nb := S.nb
$S \longrightarrow \epsilon$	écrire S.nb

1. Quel est le type de l'attribut **nb** (hérité ou synthétisé)
2. Quel est le type de cette DDS (S-attribuée ou L-attribuée) ? Et pourquoi ?
3. Donnez l'arbre de dérivation décoré pour la chaine d'entrée (**( )) ( )**

Contrôle de type

La phase d'analyse sémantique comprend un **contrôle de type** qui vise à vérifier si les opérandes de chaque operateur respectent les spécifications du langage utilisé dans le code source.

- L'objectif consiste à vérifier le sens du code source, qui est déterminé par la sémantique du langage.
- L'analyse sémantique a pour objectif de générer un arbre syntaxique abstraite qui soit sémantiquement valide.

Sous tâches de l'analyseur sémantique

1- Remplissage et consultation de la table des symboles

- La déclaration de toute variable/fonction utilisée.
- Chaque fonction est appelée en utilisant le nombre adéquat de paramètres.

2- Application des règles du système de typage

- Compatibilité entre types déclarés et utilisations des variables
- Compatibilité entre opérateurs et opérandes dans expressions
- Compatibilité entre paramètres réels et déclarés des fonctions
- Compatibilité entre valeur retournée et type de la fonction déclarée
- Conversions et coercitions automatiques

Exemple

- En analysant sémantiquement « a=b+2*c ; » implique que si a est de type entier, b, c le sont également, sinon il est nécessaire d'indiquer une erreur.
- Lorsque a, b, c sont des valeurs réelles, l'analyseur sémantique devra effectuer une opération de conversion de type pour convertir la valeur entier 2 en valeur réelle 2.0.
- Un contrôle de type **statique** est un contrôle de type effectué lors de la compilation, tandis qu'un contrôle de type **dynamique** est un contrôle de type effectué lors de l'exécution du programme cible.
- Il est préférable de ne pas utiliser de contrôle dynamique car cela rend très difficile pour le programmeur de déterminer la source de l'erreur. Toutefois, dans la réalité, certains contrôles ne peuvent être effectués que de manière dynamique.

Règle de production	action sémantique	
$I \rightarrow Id = E$	I.type :=	si Id.type=E.type alors vide 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,...)

Exemple d'un TDS de contrôle de type

Environnement d’exécution

1. Les objets statiques

- Il existe des objets statiques tout au long de l'exécution d'un programme. : leur emplacement est attribué par le compilateur lors de la compilation.
- Les fonctions, les constantes et les variables globales sont des objets statiques. Les objets statiques ont la possibilité de lire seuls ou en lecture écriture. Les fonctions et les constantes ne peuvent être liées qu'en lecture seule. Les variables globales sont fait en lecture écriture.

Espace statique *Définition*

L'espace statique désigne l'**espace mémoire** où les objets statiques sont stockés. Il se compose habituellement de deux zones : la zone du code, où se trouvent les fonctions et les constantes, et l'espace global, où se trouvent les variables globales.

L’adresse d’un objet statique *Définition*

Cela correspond à **un nombre entier** qui représente la première cellule de la mémoire occupée par l'objet. Elle est généralement exprimée comme un décalage par rapport au début de la zone où se trouve l'objet en question.

2. Les objets automatiques

Sont les variables locales des fonctions, ce qui comprend :

- Les variables déclarées à l’intérieur des fonctions,
- Les arguments formels de ces dernières.

Ces variables se trouvent dans un espace qui n'existe pas tout au long de l'exécution du programme, mais seulement lorsqu'il est nécessaire. Lorsqu'une fonction est activée, elle débute en allouant son espace local de manière adéquate (taille suffisante) pour stocker ses arguments et ses variables locales.

Chaque fois qu'une fonction est appelée, un nouvel espace local est attribué, même si cette fonction était déjà active et donc qu'un espace local pour elle existait déjà. Un espace local est détruit lorsque l'activation d'une fonction se termine.

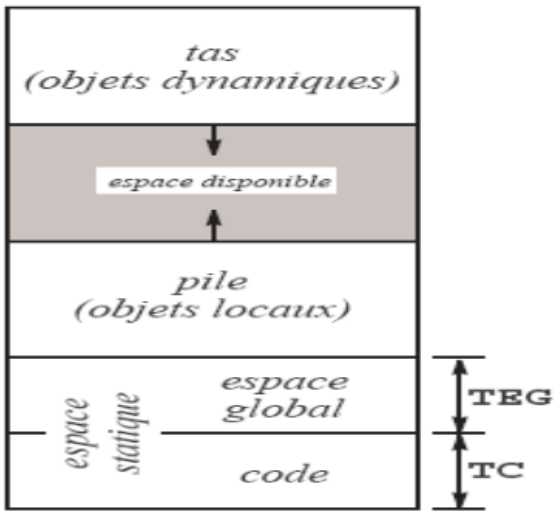
Remarque

La fonction qui se termine est la plus récente de celles qui ont été mises en place et ne sont pas encore terminée . Cela signifie que les espaces locaux des fonctions peuvent être attribués dans une mémoire gérée telle qu'une **pile**, c'est-à-dire que lorsque la fonction est activée, son espace local est créé au-

dessus des espaces locaux des fonctions actives. Ensuite, lorsque la fonction se termine, son espace local est celui située au sommet de la pile, et il suffit de le dépiler pour obtenir l'espace local de la fonction qui retournera au contrôle.

3. Les objets dynamiques

- Ils sont allouée à la demande explicite du programme (par exemple en utilisant la fonction malloc de C). Si leur destruction n'est pas spécifiquement requise, ces objets restent disponibles jusqu'à la fin du programme.
- Il est impossible de loger les objets dynamiques dans les zones où se trouvent les objets statiques, car leur existence est incertaine et dépend des conditions d'exécution. Ils ne peuvent plus être logés dans la pile des espaces locaux, car cette pile évolue et diminue en fonction des appels et des retours des fonctions, tandis que les moments de la création et de la destruction des objets dynamiques ne sont pas a priori connus. Les objets dynamiques sont alors placés dans un troisième espace, nommé le tas (heap, en anglais).
- La gestion du tas consiste à l'attribution de fragments de tailles différentes à la demande du programme, et à la récupération des fragments rendus par ce dernier.
- Le bloc de mémoire réservé à la mise en œuvre par le système d'exploitation cible est divisé en 4 zones :



Le bloc de mémoire allouées à l'exécution

- Le *code* (espace statique en lecture seule), contenant le programme et les constantes,
- L'*espace global* (espace statique en lecture-écriture), contenant les variables globales,
- La *pile* (*stack*), contenant variables locales *[1]*[1],
- Le *tas* (*heap*), contenant les variables allouées dynamiquement.

Génération du code

1. Les processeurs

Tous les processeurs sont similaires du point de vue de l'utilisateur (c'est-à-dire de la partie visible de leur architecture).Ils sont principalement constitués d'une mémoire, d'un ensemble de registres et d'un jeu d'instructions .

Les différences proviennent surtout du jeu d'instructions:

- Les (anciens) processeurs CISC (Complex Instruction Set) sont : La taille de leurs instructions varie et la plupart effectuent des transferts avec la mémoire ; ils ont généralement peu de registres (et ne sont pas uniformes). Fabriqués avant 1985. En général, Intel 8086 et Motorola 68000.
- Les (nouveaux) processeurs RISC (Reduced Instruction Set) : Les instructions sont de taille fixe et régulières (trois adresses) et elles font peu de transferts avec la mémoire. En général, ils disposent d'un grand nombre de registres (uniformes). Élaborés après 1985. En général, Alpha, Sparc et Mips.

Le langage assembleur donne aux instructions, aux registres et aux adresses constantes des noms symboliques.

L'architecture externe du MIPS (Microprocessor without Interlocked Pipeline Stages) est un processeur industriel de 32 bits, développé dans les années 80, qui représente ce qu'un programmeur doit connaître pour programmer en assembleur.

Le microprocesseur MIPS 32 bits est composé de plusieurs registres. Les registres sont de petites mémoires extrêmement rapides qui conservent des informations (il est important de se rappeler que nous disposons de nombreuses mémoires : mémoire secondaire, mémoire RAM et mémoire cache).

L'assemblage, également appelé langage assembleur, vous permet d'avoir davantage de contrôle que C++ et Java parce qu'il est en bas niveau.

2. Règle de génération du code MIPS

Les registres de Mips *Fondamental*

Le processeur MIPS dispose de 32 registres de travail qui sont accessibles au programmeur. Chaque registre est identifié par un numéro entre 0 à 31 et est représenté par un $.

- $zero (son numéro est : 0) : c'est le registre zéro qui stocke l'instante zéro.
- $at (son numéro est : 1) c'est le registre assembly temporary (réservé à l'assembleur).
- $v0 et $v1 (leurs numéros sont : 2 et 3) retournent des résultats des variables et d'évaluation des expressions.
- $a0 - $a3 (leurs numéros sont : 4, 5, 6 et 7) sont des registres d'argument des fonctions.
- $t0- $t7 (leurs numéros sont : 8, 9, 10, 11, 12, 13, 14 et 15) stockent des informations temporaires.

- $s0 - $s7 (leurs numéros sont : 16, 17, 18, 19, 20,21, 22 et 23) vous permettent de sauvegarder des informations des appels.
- $t8- $t9 (leurs numéros sont : 24 et 25) sont des registres temporaires.
- $k0- $k1 (leurs numéros sont :26 et 27) sont réservés pour le kernel (système d'exploitation).
- $gp (son numéro est : 28) global pointer registre de pointeur global.
- $sp (son numéro est : 29) stack pointer registre de pointeur de pile.
- $fp (son numéro est : 30) frame pointer registre de pointeur d'image.
- $ra (son numéro est : 31) return adresse registre de retour d'adresse.

Les modes d'adressage ***Fondamental***

- L'architecture MIPS est de type chargement/rangement, ce qui implique que seules les instructions de chargement et de rangement peuvent accéder à la mémoire. Les instructions de calcul ne fonctionnent que sur des données existent dans des registres. MIPS n'a qu'une seule méthode d'adressage pour lire ou écrire des données en mémoire: l'adressage indirect registre avec déplacement : **I(rx)**. L'adresse est obtenue en additionnant le déplacement I (positif ou négatif) au contenu du registre **rx**.
- ***Exemples:*** lw $12, 13($10) # $12 <= Mem[$10 + 13] **chargement (load)**
  sw $20, -60($22) # Mem[$22-60] <= $20 **rangement (store)**

Directives de l'assembleur ***Fondamental***

Le terme "directives" désigne les commandes fournies à l'assembleur afin de structurer les programmes. Toutes les directives commencent par un point (.). La liste des différentes directives est la suivante :

1. .data :dans cette directive, l'assembleur est informé que la section suivante est celle de la déclaration des données, en réalité, c'est l'adresse du segment de données (0x1001 0000).
2. .text : selon cette directive, la section suivante est celle consacrée aux instructions du programme. En fait, c'est l'adresse du segment du programme (0x00400000).
3. .byte , .half , .word :les 3 directives autorisent la déclaration des entiers signés dans le segment de données, avec des tailles respectives de 8 bits, 16 bits et 32 bits.
4. .float et .double :les deux directives sont identiques aux types réels de même nom dans le langage C++, à savoir float et double.
5. .ascii et .asciiz : les directives de déclaration de données sont également utilisées pour déclarer des chaînes de caractères avec un encodage en ASCII, les chaînes de caractères étant ainsi déclarées comme des tableaux de caractères. Le format **.asciiz** diffère de.**ascii** en ce qu'il nécessite un caractère null (0x00 ou \0 en C++) à la fin de la chaîne de caractères (comme dans le langage C) afin de reconnaître la fin de la chaîne.

Format d'un programme MIPS : chaque programme assembleur MIPS R3000 doit avoir le format suivant :

```
.data
#---------------
#---------------
.text
```

#----------

#----------

Instructions

Fondamental

- Selon les informations suivantes, le registre noté **$rr** est le registre de destination, tandis que les registres notés **$ri** et **$rj** sont les registres de source qui constituent les valeurs des opérandes sur lesquelles l'opération est effectuée. Il convient de souligner que le registre source peut être le registre de destination d'une même instruction assembleur. Un opérande immédiat sera indiqué comme **imm**, et sa taille sera précisée dans la description de l'instruction.
- Les instructions de saut utilisent une étiquette (ou label) comme argument, qui permet de déterminer l'adresse de saut. Chaque instruction entraîne une modification du registre **PC**, qui inclut l'adresse de l'instruction suivante à exécuter. Finalement, le résultat d'une multiplication ou d'une division est classé dans deux registres spécifiques, **HI** pour les poids forts et **LO** pour les poids faibles. Cela nous conduit à présenter quelques notations :

+	Addition entière en complément à 2
-	Soustraction entière en complément à 2
x	Multiplication entière en complément à 2
/	Division entière en complément à 2
mod	Reste de la division entière en complément à 2
and	Opérateur et logique bit à bit
or	Opérateur ou logique bit à bit
nor	Opérateur non-ou logique bit à bit
xor	Opérateur ou-exclusif logique bit à bit
Mem[ad]	Contenu de la mémoire à l'adresse ad
<=	Assignation
||	Concaténation entre deux chaînes de bits
B^n	Réplication du bit B n fois dans une chaîne de bits
$X_{p...q}$	Sélection des bits p à q dans une chaîne de bits X

Macro-instructions

Fondamental

Une macro-instruction est une pseudo-instruction qui n'est pas incluse dans le jeu d'instructions machine, mais qui est acceptée par l'assembleur qui la transforme en une séquence de plusieurs instructions machine. Le registre **$1** est utilisé par les macro-instructions lorsqu'elles nécessitent un calcul intermédiaire.Il est donc déconseillé d'utiliser ce registre dans les programmes.

la

Chargement d'une adresse dans un registre

Syntaxe : la $rr, adr

Description : L'adresse considérée comme une quantité non-signée est chargée dans le registre.

li

Chargement d'un opérande immédiat sur 32 bits dans un registre.

Syntaxe : li $rr, imm

Description : La valeur immédiate est chargée dans le registre $rr .

Appels systeme *Fondamental*

- Le programme utilisateur doit utiliser un "appel système" l'instruction **syscall** pour effectuer certains traitements qui ne peuvent être effectués que sous le contrôle du système d'exploitation (généralement les entrées/sorties qui impliquent la lecture ou l'écriture d'un nombre ou d'une chaîne de caractères sur la console).
- En règle générale, le numéro de l'appel système est inclus dans le registre **$2**, tandis que ses arguments potentiels sont inclus dans les registres **$4 et $5**. Il y'a cinq appels système :

Fondamental

- **Écrire un entier**

Pour écrire (afficher) un entier, il faut :

1. Mettre l'entier à écrire dans le registre $4.
2. Mettre la valeur 1 dans le $2.
3. syscall

Exemple

```
li      $4,     0x1234567     # stocke la valeur hexadécimale 1234567 dans $4
ori     $2,     $0,     1     # code de « print_integer » dans $2
syscall                       # affiche 1234567
```

Fondamental

- **Lire un entier sur la console**

Lire un entier consiste à exécuter l'appel système numéro 5 et récupérer le résultat dans le registre $2.

Exemple

```
ori     $2,     $0,     5     # code de « read_integer » dans $2
syscall                       # $2 contient la valeur lue
```

Fondamental

- **Ecrire une chaîne de caractères**

Pour écrire une chaîne de caractères, il faut :

1. Mettre l'adresse de la chaîne à afficher dans le $4.
2. Mettre la valeur 4 dans le $2
3. syscall

Exemple

```
.data
str:    .asciiz "Chaîne à afficher\n"
```

```
.text
la      $4,     str             # charge l'adresse de la chaîne dans $4
ori     $2,     $0,     4       # code de « print_string » dans $2
syscall                         # affiche la chaîne pointée
```

Fondamental

- **Lire une chaîne de caractères sur la console**

Pour lire une chaîne de caractères, il faut un pointeur définissant l'adresse du buffer de réception en mémoire et un entier définissant la taille du buffer (en nombre de caractères).

On écrit la valeur du pointeur dans $4, et la taille du buffer dans $5, et on exécute l'appel système numéro 8.

Exemple

```
read:   .space 256
la      $4,     read            # charge le pointeur dans $4
ori     $5,     $0,     152     # charge longueur max dans $5
ori     2,      $0,     8       # code de « read_string »
syscall                         # copie la chaîne dans le buffer pointé par $4
```

Fondamental

- **Terminer un programme**

Pour terminer un programme, il faut exécuter l'appel système numéro 10. C'est à dire :

1. Mettre la valeur 10 dans le $2
2. syscall

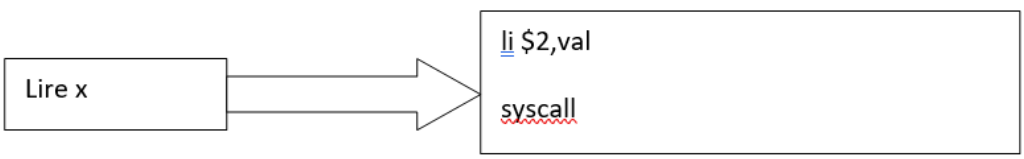
Exemple

```
ori     $2,     $0,     10      # code de « exit »
syscall                         # quitte pour de bon
```

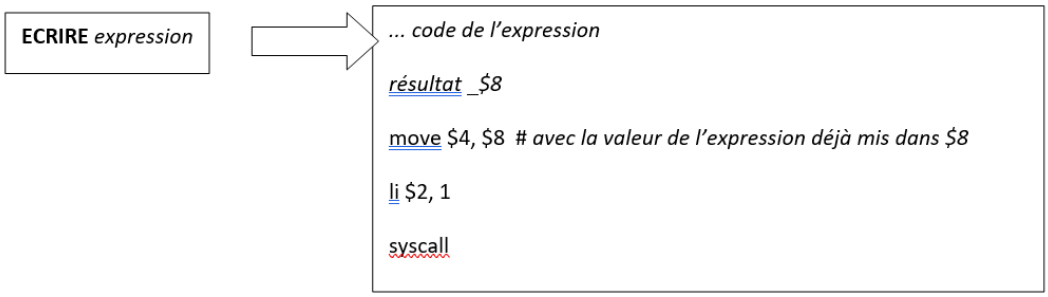
3. Quelques exemples de génération du code Mips

Fondamental

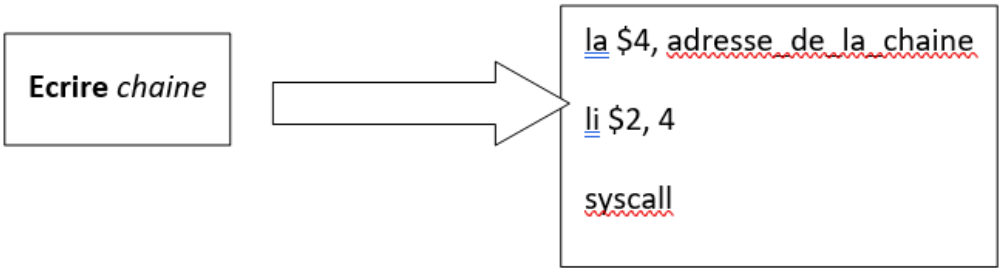
- **Lecture**



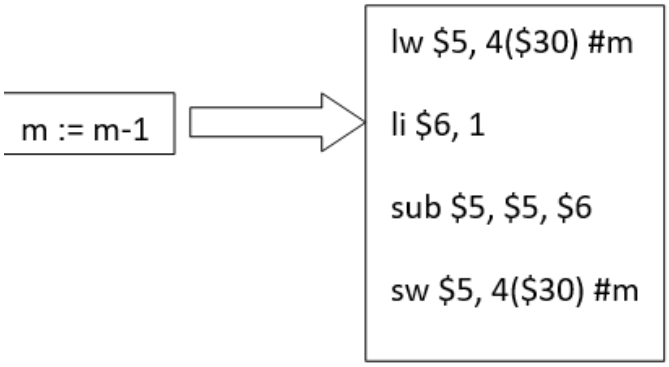
- **Ecriture**



- **Chaînes de caractères**

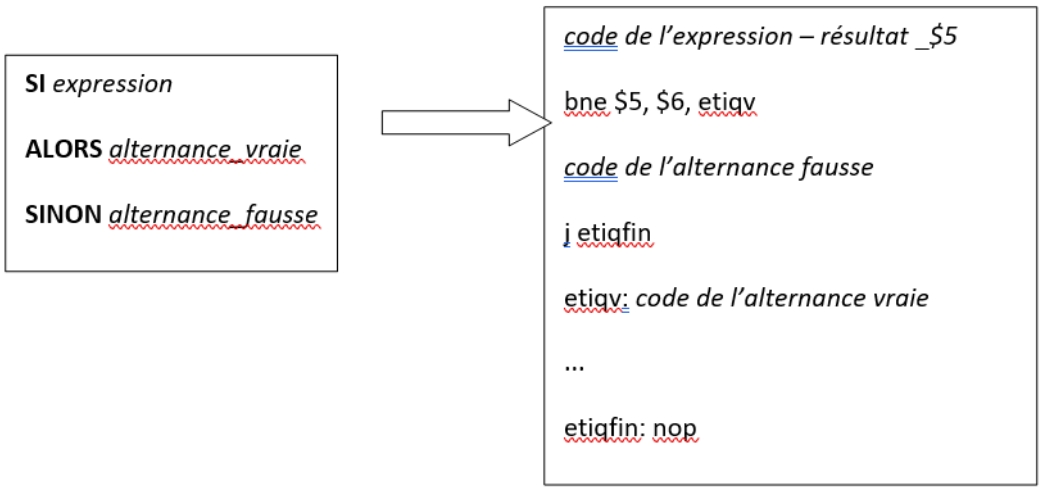


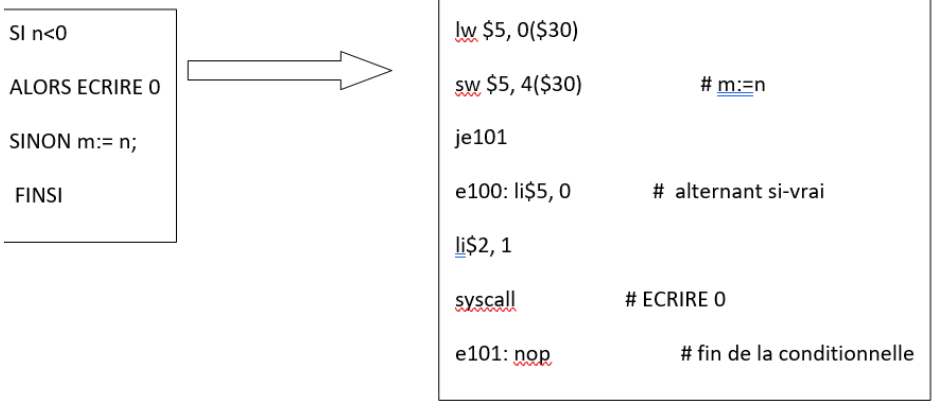
- **Affectation**
  - D'abord une séquence de code calculant la valeur de l'expression et la logeant dans un registre fixe (reg. de base $5)
  - Ensuite (sans étiquette) un transfert "store" $5



- **Conditionnel**

On génère 2 étiquettes ***etiqv***, ***etiqfin***

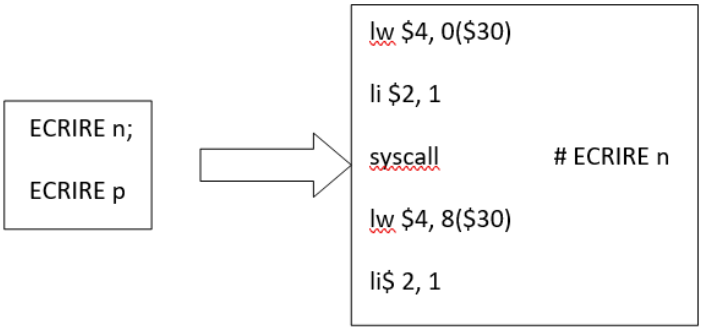




- **Composition**

La composition des instructions se traduit en assembleur par la simple mise en séquence

Exemple : soit la composition de deux instructions d'écriture :



- **Comparaisons**

1. Séquence de code évaluant exp_ar1, avec valeur dans le registre de base $5
2. Séquence de code (sans étiquette) évaluant exp_ar2 , avec valeur dans le registre suivant $6
3. Branchement conditionnel (dépendant du relateur, et sans étiquette) sur $5 et $6, vers une étiquette "vrai" bcond $5,$6, vrai

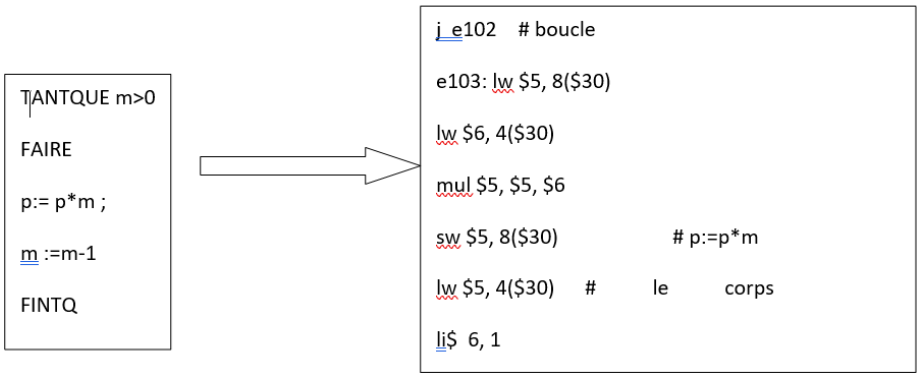
Exemple

etiq:lw $5, 0($30) #n valeur du premier terme dans $5

li $6, 0 #valeur du second terme dans $6

blt $5, $6,e100 saut si $5 < $6

- **Boucles**



- **Expressions arithmétiques**

1. Séquence évaluant l'opérande gauche, avec valeur dans $k
2. Séquence (sans étiquette) évaluant l'opérande droit, avec valeur dans $k+1
3. Instruction arithmétique (sans étiquette) opérant sur $k, $k+1 et plaçant le résultat dans $k et l'instruction *op $k, $k,$k+1*

Exemple

Code engendré pour l'affectation **n := n + 2 + m/p** avec la valeur passant par $5

```
lw $5, 0($30)
li$ 6, 2
add $5, $5, $6
lw $6, 4($30)
lw $7, 8($30)
div $6, $6,$7
add $5, $5, $6
sw $5, 0($30)
```

4. Exercices

4.1. Exercice 01

L'algorithme de la factorielle en C++ est comme suivant:

```
int n , F ;
cout << "Donnez S.V.P. un nombre positive: " ;
cin >> n ;
F = 1 ;
while(n > 0)
{
F = F * n ;
n-- ;
}
cout << F << "La factorielle est : " << endl ;
```

1. Faire le programme de la factorielle en MIPS

4.2. Exercice 02

Écrire un programme qui prend en argument (sur la ligne de commande) une suite de valeurs entières ou réelles, avec au plus 5 valeurs de chaque type, et qui produit en sortie un programme en assembleur MIPS qui affiche tous les entiers sur une première ligne et tous les réels sur une deuxième ligne.

Par exemple, pour la suite 5 4.56 8.6 12 0.25 8 8, le programme produira en sortie le programme assembleur ci-dessous dont l'exécution affichera :

```
5 12 8 8
4.56 8.6 0.25
```

4.3. Exercice 03

1- On suppose que l'adresse du début du bloc réservé des données est mise dans le registre $30 et que la variable a (resp. b, c, x et continue) ont respectivement leur place décalée de 0(resp.4,8,12 et16) octet(s) de cette adresse. On remarque un décalage de 4 octets (32bits) par variable.

2- Ecrire dans un pseudo langage un programme permettant de décaler 5 variables (a,b,c,x et continue), de lire a, b, c, et x et de calculer et afficher la valeur de c+bx+ax2.

3- Pour évaluer cette expression arithmétique, on propose d'abord de construire son arbre de syntaxe abstraite.

4- Ecrire les codes MIPS permettant de lire a,b,c, et x, puis d'évaluer l'expression en faisant le parcours gauche-droite et d'afficher le résultat.

5- On souhaite pouvoir poser la question à l'utilisateur qui répondra par un non (entier 0) ou par un oui (entier non nul) pour reprendre les saisies et le calcul de la question 1.

6- Pour calculer la suite de Fibonacci définie de la manière suivante :

U0=0

U1=1

Un=Un-1+Un-2

On propose l'algorithme suivante :

Si n==0

ALORS ECRIRE 0

SINON SI n==1

ALORS ECRIRE 1

SINON {

Un2 :=0 ;

Un1 :=1 ;

TANTQUE n>=2 FAIRE

{

U :=Un1 ;

Un1 :=U+ Un2 ;

Un2 :=U ;

n :=n-1

}

ECRIRE "le nombre de Fibonacci de n est =\n " ;

ECRIRE Un1

}

Construire l'arbre correspondant et donner le code MIPS complet pour ce programme

Références

1. Mohamed BOUHDADI, « Compilation-théorie de langages » ,Université Mohammed V - Agdal,Faculté des Sciences, Rabat
2. Habib ABDULRAB,Claude MOULIN ,Sid TOUATI « Compilation : théorie, techniques et outils » ,Université de technologie de compiègne
3. Meadi M.Nadjib , « Support de cours Compilation » ,Université de Biskra
4. Alfred AHO,Jeffrey ULLMAN, « Compilers: Principles, Techniques, and Tools », First Edition (1986)