الجمهورية الجزائرية الديمقراطية الشعبية

République Algérienne Démocratique et Populaire

وزارة التعليم العالي و البحث العلمي

Ministère de L'Enseignement Supérieur et de la Recherche Scientifique

N°Ref :……

Centre Universitaire Abdelhafid BOUSSOUF- Mila

Institut de Mathématiques et Informatique

Département Informatique

Publication pédagogique

Semestre :5

Parcours SI

Unité d'enseignement fondamentale : UEF1

Matière : Compilation

Crédits :5

Coefficient : 3

Préparé par :

Hettab Abdelkamel

Année Universitaire : 2024/2025

Table des matières

Chapitre 1. Introduction aux compilateurs (Objectifs)

Chapitre 2. Compilation (Introduction)

Chapitre 3. Analyse lexicale

Chapitre 6. Analyse syntaxique ascendante

Table des matières

Chapitre 1. Introduction aux compilateurs (Objectifs)

1.1 Introduction :

La compilation est une étape cruciale dans le processus de développement logiciel. Elle consiste à traduire un code source, écrit dans un langage de programmation de haut niveau, en un code binaire ou exécutable que la machine peut comprendre et exécuter. Ce processus est assuré par un compilateur, qui agit comme un traducteur entre le programmeur et l'ordinateur.

Un compilateur effectue plusieurs tâches essentielles, telles que l'analyse du code source, l'optimisation des instructions, et la génération d'un code cible efficace. Il contribue à garantir que le programme fonctionne correctement tout en maximisant ses performances.

1.2 Principe de la compilation :

La compilation est le processus de transformation d'un programme écrit dans un langage de haut niveau (compréhensible par les humains) en un langage de bas niveau ou un code machine (exécutable par l'ordinateur). Ce processus repose sur une série d'étapes, chacune ayant un rôle spécifique. Voici les principes clés de la compilation :

- **Analyse du programme source**

Cette étape consiste à comprendre le programme écrit par le développeur. Elle se décompose en plusieurs phases :

  - Analyse lexicale
  - Analyse syntaxique
  - Analyse sémantique

- **Transformation intermédiaire :**

Le programme est traduit en une représentation intermédiaire, indépendante de la machine cible.

- **Optimisation du code :**

Amélioration de l'efficacité du programme, en réduisant le nombre d'instructions ou l'utilisation de ressources (mémoire, registres).

- **Génération de code :**

Traduction de la représentation intermédiaire en un code machine spécifique à l'architecture cible.

- **Assemblage et édition des liens (Linking) :**

- Assemblage : Traduction du code machine en langage binaire exécutable.
- Linking : Combinaison des différents fichiers objets (produits par le compilateur) pour créer un exécutable complet.
- Gestion des bibliothèques statiques et dynamiques.

1.3 Objectifs de la compilation :

La compilation est un pilier fondamental du cycle de vie d'un logiciel. Elle garantit que les intentions du développeur sont traduites en instructions que la machine peut exécuter de manière fiable et performante. Les objectifs principaux du compilateur sont :

- **Traduction du code source** :
  - Convertir un langage lisible par l'humain (C, Java, Python, etc.) en langage machine ou intermédiaire.
  - Produire un binaire ou un fichier exécutable qui peut être directement exécuté par le processeur.
- **Détection des erreurs** : Un compilateur doit pouvoir détecter les erreurs statiques comme :
  - Identificateurs mal formés,
  - Commentaires non fermés . . .
  - Constructions syntaxiques incorrectes,
  - Identificateurs non déclarés,
  - Expressions mal typées: **ex :** if 3 then "toto" else 4.5,
  - Références non instanciées.
- **Optimisation des performances** :
  - Générer un code machine efficace qui utilise de manière optimale les ressources matérielles (mémoire, processeur, etc.).
  - Réduire le temps d'exécution et l'utilisation des ressources.
- **Portabilité** :
  - Permettre à un programme écrit dans un langage de haut niveau d'être compilé pour différentes architectures matérielles et systèmes d'exploitation.
  - Assurer une abstraction entre le code source et la plateforme cible.
- **Automatisation des tâches complexes** :
  - Simplifier le processus de gestion des dépendances et des bibliothèques.

- Intégrer des fonctionnalités avancées, comme la gestion de la mémoire ou le traitement parallèle.

- **Facilitation de la maintenance** :
  - Fournir des outils, comme des messages d'avertissement ou de diagnostic, qui aident à maintenir et à améliorer le code source.

1.4 Qu'attend-on d'un compilateur ?

Les trois aspects suivants : efficacité, correction et modalité sont très essentiel pour refléter les exigences nécessaires pour un compilateur moderne, en particulier dans le cadre de projets complexes. Ils garantissent un développement fluide, des performances élevées et une maintenance simplifiée.

1.4.1 Efficacité :

Un compilateur doit être rapide afin de minimiser le temps nécessaire à la compilation d'un programme. Cela est particulièrement important dans des projets de grande envergure où des centaines ou des milliers de fichiers doivent être compilés régulièrement. L'efficacité d'un compilateur peut être mesurée selon plusieurs aspects :

- **Temps de compilation** : Un compilateur performant réduit le délai entre l'écriture du code et son exécution, ce qui accélère le cycle de développement.
- **Efficacité du code généré** : Le compilateur doit produire un code machine optimisé, qui s'exécute rapidement tout en utilisant efficacement les ressources matérielles (CPU, mémoire, etc.).

1.4.2 Correction :

Un critère fondamental pour un compilateur est qu'il doit garantir que le code compilé correspond fidèlement au comportement prévu par le programme source :

- **Préservation de la logique** : Le programme résultant doit effectuer exactement les calculs ou opérations définis dans le code source.
- **Respect des normes** : Le compilateur doit suivre les spécifications du langage pour éviter tout comportement indéfini ou incorrect.
- **Signalement des erreurs** : Il est également de la responsabilité du compilateur de détecter les erreurs syntaxiques ou sémantiques dans le programme original, afin de les corriger avant que le code ne soit exécuté.

1.4.3 Modalité : Compilation séparée :

Lors de grands développements logiciels, il devient impraticable de recompiler l'intégralité du projet à chaque modification. La **compilation séparée** offre une solution à ce problème :

- **Principe** : La compilation séparée consiste à diviser le programme en plusieurs unités ou modules (fichiers source). Chaque module peut être compilé individuellement, produisant des fichiers objet intermédiaires.
- **Avantages** :
    - Réduction du temps de compilation : Seuls les modules modifiés sont recompilés.
    - Meilleure organisation : Permet une gestion plus claire et modulaire des grandes bases de code.
    - Facilitation du travail en équipe : Chaque développeur peut travailler sur un module spécifique sans interférer avec les autres.
- **Processus** :

1. Chaque module est compilé indépendamment.
2. Les fichiers objets générés sont ensuite liés (linkés) pour produire le programme final.

1.5 Pourquoi étudier la compilation ?

Étudier la compilation offre de nombreux avantages, tant sur le plan théorique que pratique, qui contribuent à améliorer les compétences en informatique et à élargir les perspectives professionnelles. Voici les principales raisons :

1.5.1 Ça vous rendra plus compétent :

La compilation exige de comprendre des concepts avancés, tels que les **structures de données**, les **algorithmes**, la **théorie des langages**, et les **architectures matérielles**.

Cette étude approfondit la capacité à résoudre des problèmes complexes et renforce la maîtrise des bases fondamentales en informatique.

1.5.2 Les techniques de compilation sont utilisées partout :

Les concepts de compilation (analyse lexicale, parsing, optimisation) ne se limitent pas aux compilateurs :

- **Interprètes** (ex. Python, JavaScript).
- **Bases de données** (optimisation des requêtes SQL).
- **Langages embarqués** pour le développement de logiciels système.
- **Machines virtuelles** (Java Virtual Machine, .NET).

Les techniques apprises sont également applicables dans d'autres domaines, comme l'analyse statique de code ou les outils de transformation de texte.

1.5.3 Ce sont des systèmes techniquement intéressants :

Un compilateur est un exemple fascinant d'un **système logiciel complexe** où interagissent plusieurs domaines :

- Théorie des langages.
- Génie logiciel.
- Architecture des ordinateurs.

L'étude d'un compilateur offre une vue d'ensemble unique sur la conception et la mise en œuvre de logiciels sophistiqués.

1.5.4 Pour devenir de meilleurs programmeurs :

Comprendre comment les langages de programmation fonctionnent **sous le capot** améliore les compétences de codage :

- Écrire du code plus efficace et adapté à l'architecture cible.
- Apprendre à éviter des erreurs courantes, comme les erreurs de typage ou d'allocation de mémoire.

Cela inculque des pratiques de conception robustes, comme l'évaluation des dépendances entre les instructions.

1.5.5 Pour faire croître sa boîte à outils :

L'étude de la compilation enrichit vos connaissances et vous donne des outils puissants :

- **Expressions régulières**, automates, grammaires.
- Génération de code intermédiaire et optimisation.
- Familiarité avec des outils comme **Lex**, **YACC**, ou **LLVM**.

Ces compétences sont précieuses non seulement pour développer des compilateurs, mais aussi pour créer des outils d'analyse de code, des traducteurs de formats, ou encore des interpréteurs.

1.6 Contenu de la matière :

Chapitre 1 : Introduction (Objectif)

- Présentation des objectifs de l'étude de la compilation.
- Explication des enjeux liés à la traduction de langages de haut niveau en code machine.
- Importance des compilateurs dans le cycle de développement logiciel.

Chapitre 2 : Compilation

Chapitre 1. Introduction aux compilateurs (Objectifs)

i. **Définition d'un compilateur** :

- Description du rôle d'un compilateur dans le processus de développement logiciel.
- Différenciation entre compilation et interprétation.

ii. **Structure d'un compilateur** :

- Les différentes étapes de la compilation : analyse lexicale, syntaxique, sémantique, génération de code intermédiaire, optimisation, et génération de code.
- Illustration d'un compilateur sous forme de diagramme.

Chapitre 3 : Analyse lexicale

- Identification des unités lexicales (tokens) dans un programme source.
- Utilisation des automates finis et des expressions régulières pour modéliser les lexèmes.
- Introduction aux outils pratiques tels que **Lex** pour générer des analyseurs lexicaux.

Chapitre 4 : Analyse syntaxique

i. **Dérivation la plus à gauche et arbre de dérivation** :

- Construction des dérivations et des arbres syntaxiques.

ii. **Grammaire ambiguë** :

- Définition et exemples de grammaires ambiguës.
- Conséquences pour l'analyse syntaxique.

iii. **Grammaire et langages de programmation** :

- Relation entre les grammaires formelles et les langages de programmation.
- Importance des grammaires contextuelles.

iv. **Analyseurs syntaxiques et leurs types** :

- Classification des analyseurs syntaxiques : analyse descendante et analyse ascendante.

v. **Outils en pratique** :

- Introduction aux outils comme **YACC** et **AntLR** pour créer des analyseurs syntaxiques.

Chapitre 5 : Analyse descendante

i. **Analyse LL(1) (principe)** :

- Fonctionnement des analyseurs LL(1).

ii. **Table d'analyse** :

- Construction et utilisation de la table d'analyse LL(1).

iii. **Grammaire LL(1)** :

- Identification et transformation des grammaires adaptées pour l'analyse LL(1).

Chapitre 6 : Analyse ascendante

i. **Analyse LR (principe)** :

- Introduction générale aux méthodes d'analyse ascendante.

ii. **Analyse LR(0)** :

- Fonctionnement des analyseurs LR(0) et leurs limitations.

iii. **Analyse SLR(1)** :

- Améliorations apportées par les analyseurs SLR(1).

iv. **Analyse LR(1)** :

- Description des analyseurs LR(1) et de leur puissance expressive.

v. **Analyse LALR(1)** :

- Présentation des analyseurs LALR(1), un compromis entre SLR(1) et LR(1).

Chapitre 7 : Traduction dirigée par la syntaxe

- Introduction aux notions de S-attribuées et L-attribuées.
- Utilisation des règles sémantiques pour extraire des informations sur le programme source.
- Cas d'utilisation pour traduire des instructions en représentations intermédiaires.

Chapitre 8 : Contrôle de type

- Vérification et analyse des types dans un programme.
- Détection des incompatibilités de types.
- Exemples de systèmes de typage et leur intégration dans un compilateur.

Chapitre 9 : Environnement d'exécution

- Gestion de la mémoire et des structures de données lors de l'exécution d'un programme.
- Allocation, gestion des piles d'appels et des tas.
- Stratégies de passage des paramètres (par valeur, par référence).

Chapitre 10 : Génération de code

- Production de code machine ou intermédiaire à partir du programme source.
- Génération de code pour des instructions arithmétiques, des structures de contrôle et des appels de fonctions.
- Méthodes de génération de code en une ou plusieurs passes.

Chapitre 2. Compilation (Introduction)

2.1 Qu'est-ce qu'un compilateur ?

Un compilateur est un traducteur qui permet de transformer un programme écrit dans un langage **L1** en un autre programme écrit dans un langage **L2**. Le langage **L1** est appelé le **langage source**, qui est le langage dans lequel on écrit le programme original ex: C, C++, Pascal … Le langage **L2** généré à partir du langage **L1** est appelé le langage cible (ou langage objet) ex : EXE,…

Remarque : En pratique on s'arrête souvent à un langage intermédiaire (assembleur ou encore un langage d'une machine abstraite).

2.2 Qu'est-ce qu'un programme?

Un programme est un ensemble d'opérations qui décrivent comment produire des résultats à partir des entrées. Les compilateurs rejettent des programmes qui contiennent des erreurs (erreurs statiques), et dans le cas contraire, le compilateur construit un nouveau programme (le programme objet) que la machine pourra l'exécuter sur différentes entrées.

L'exécution du code objet sur une entrée particulière peut ne pas terminer ou échouer à produire un résultat (erreur dynamique).

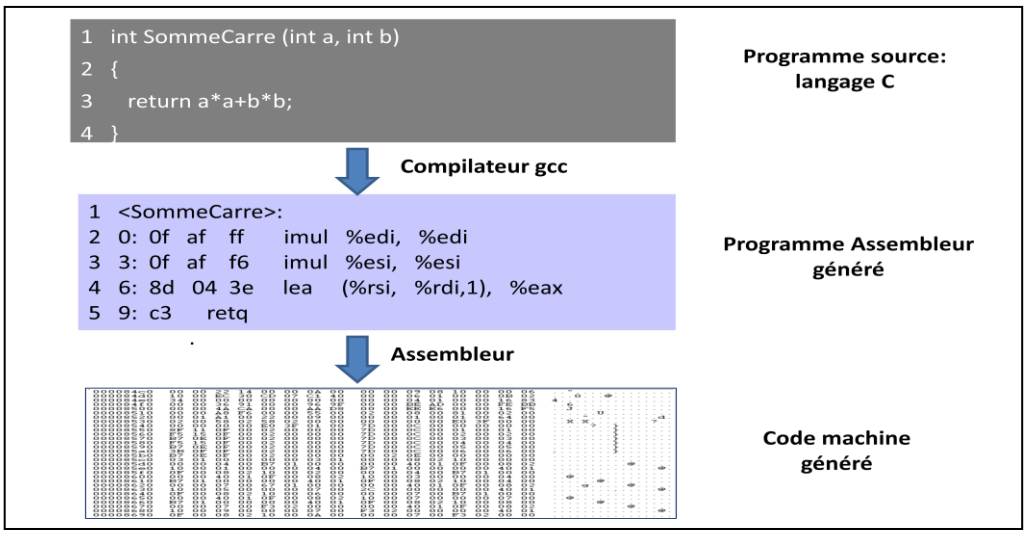


Figure 2.1 Principe de la compilation

2.3 Compilateur vs Interpréteur :

2.3.1 Compilateur :

Le **compilateur** prend tout le programme et le convertit en code objet qui est généralement stocké dans un fichier. Le code objet est également référencé en tant que code binaire et peut être exécuté directement par la machine après la liaison.

Exemple: **C, C++, Pascal** …

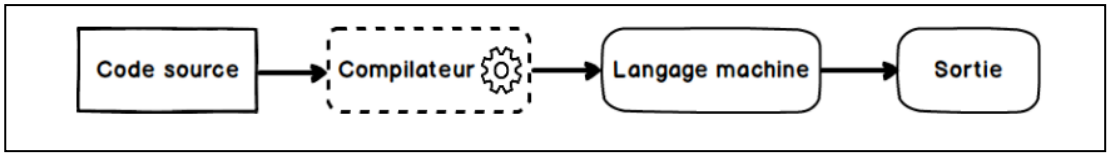


Figure 2.2 Principe du compilateur

2.3.2 Interpréteur

L'**interpréteur** exécute directement des instructions écrites dans un langage de programmation l'une après l'autre sans les convertir en un code objet ou un code machine.

Exemple: **BASIC, Perl, Python, Ruby, PHP ….**

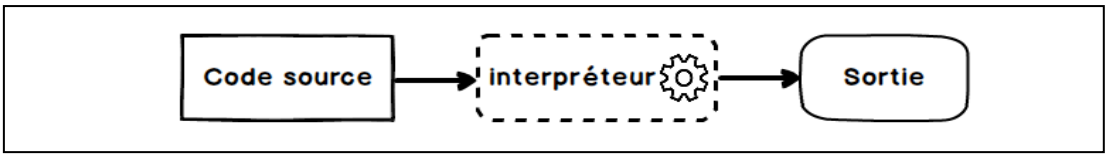


Figure 2.3 Principe d'interpréteur

2.3.3 Compilateur vs Interpréteur (Avantage et inconvénient)

Le tableau suivant montre les avantages et inconvénients des compilateurs et interpréteurs :

	Avantage	Inconvénient
Interpréteurs	processus de développement simple (en particulier débogage)	processus de traduction peu efficace et vitesse d'exécution lente
Compilateurs	Transmet au processeur l'intégralité du code machine prêt à l'emploi et exécutable	Toute modification du code nécessite une nouvelle traduction (correction des erreurs, extension du logiciel, etc.)

Tableau 2.1 avantages et inconvénients des compilateurs et interpréteurs

2.3.4 Compilateur à la volée

La compilation à la volée est une solution hybride traduit le code du programme pendant le fonctionnement, à l'instar d'un interpréteur. Elle mélange les deux méthodes précédentes pour bénéficier de leurs avantages. De cette façon, la vitesse d'exécution élevée (permise par le compilateur) est complétée par un processus de développement simplifié issus d'interpréteur.

Exemple: JAVA….

2.4 Qu'attend-on d'un compilateur ?

Un bon compilateur doit être capable de détecter des erreurs, efficace, correcte et modulaire.

- **Détection des erreurs**: Un compilateur doit pouvoir détecter les erreurs statiques comme:
  - Identificateurs mal formés,
  - commentaires non fermés . . .
  - Constructions syntaxiques incorrectes,
  - Identificateurs non déclarés,
  - Expressions mal typées : **ex :** if 3 then "toto" else 4.5,
  - Références non instanciées.
- **Efficacité** : Un compilateur doit être si possible rapide.
- **Correction** : Le programme compilé doit représenter le même calcul que le programme original.
- **Modalité** : Utilisation de la compilation séparée lors de d'un gros développement.

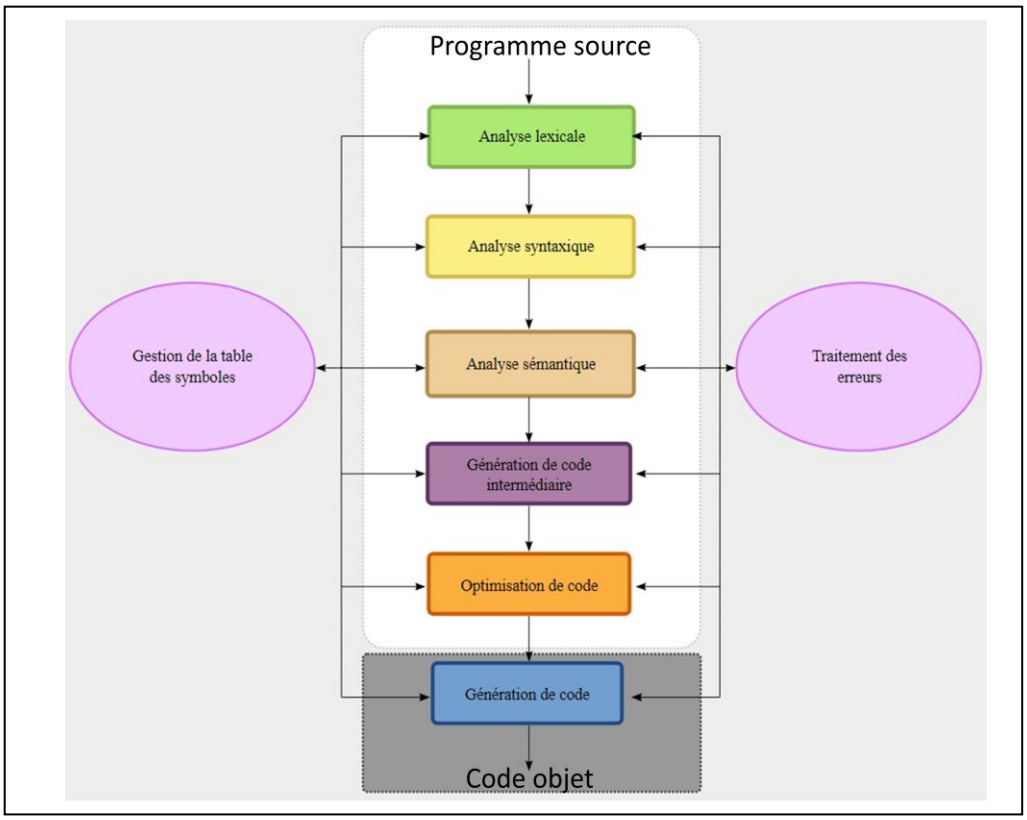


Figure 2.4 Différentes phases de la compilation

2.5 Structure d'un compilateur

En général, la plupart des compilateurs possèdent les phases suivantes : Analyse lexicale, analyse syntaxique, analyse sémantique, génération de code intermédiaire, optimisation de code intermédiaire, et génération de code objet. La figure 2.4 montre les différentes phases de la compilation.

2.5.1 Analyse lexicale

L'analyse lexicale consiste à transformer un code sous forme textuelle en une suite de tokens (unités lexicales), séparant ainsi les mots-clés, les variables, les entiers, etc....

Les commentaires et les blancs séparant les caractères formant les unités lexicales sont éliminés au cours de cette phase.

L'analyseur lexical associe à chaque unité lexicale un code qui spécifie son type et une valeur (un pointeur vers **la table des symboles**).

Exemple

Soit l'instruction Pascal suivante: **for i :=1 to vmax do a :=a+i;**

On peut dégager la suite de tokens suivante :

for : mot clé

i : identificateur

:= : affectation

1 : entier

to : mot clé

vmax : identificateur

do : mot clé

a : identificateur

:= : affectation

a : identificateur

+: opérateur arithmétique

i : identificateur

;: séparateur

2.5.1.1 Table des symboles

- Une **table de symboles** est une centralisation des informations rattachées aux unités lexicales d'un programme informatique.
- Dans une table des symboles, on retrouve des informations comme : le type, l'emplacement mémoire, etc.

- Généralement, la table est créée dynamiquement. Une première portion est créée au début de la compilation. Puis, de façon opportuniste, en fonction des besoins, elle est complétée.
- La première fois qu'un symbole est vu (au sens des règles de visibilité du langage), une entrée est créée dans la table.

Exemple

Voici la table des symboles pour l'instruction **: for i :=1 to vmax do a :=a+i;**

N°	Unité Lexicale	Type de l'unité lexicale
10	for	Mot clé
11	to	Mot clé
12	do	Mot clé
13	;	séparateur
…	……	……
100	:=	affectation
101	+	opérateur arithmétique
…	….	….
1000	i	Identificateur
1001	a	Identificateur
1002	vmax	Identificateur
….	…..	
5000	1	entier

Ensuite, l'énoncé précédent peut s'exprimer ainsi : 10, 1000, 100, 5000, 11, 1002, 12, 1001, 100, 1001, 101, 1000, 13.

2.5.2 Analyse syntaxique

- Cette phase consiste à regrouper les unités lexicales du programme source en structures grammaticales (i.e. vérifier si un programme est correctement écrit selon la grammaire qui spécifie la structure syntaxique du langage).
- En général, cette analyse est représentée par un arbre.

Exemple : Considérons la grammaire suivante pour la boucle for :

<Instruction>→ < Affectation > ;| <Instruction for>

< Affectation > → ident := < Expression>

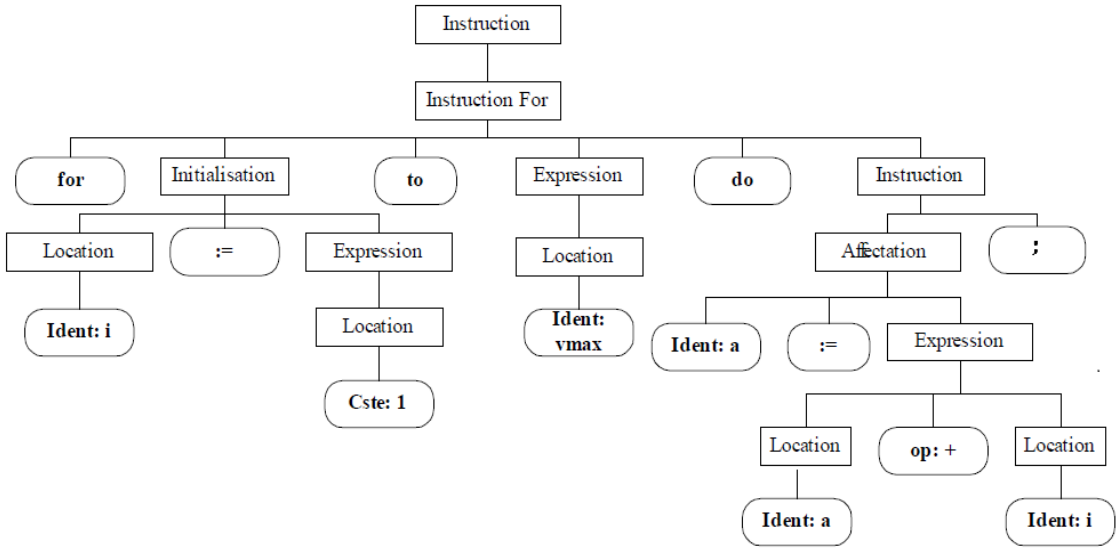
<Instruction for> → for < Initialization> to < Expression> do <Instruction>

< Initialization> → <Location> := < Expression >|<Location>

<Location> → ident|const

< Expression>→ < Location>|< Location > op < Location >

Après l'analyse syntaxique de l'instruction : **for i :=1 to vmax do a :=a+i;** on obtient l'arbre suivant :



2.5.3 Analyse sémantique

- L'analyse sémantique sert à préciser la nature des calculs représentés par un programme en vérifiant que les opérandes de chaque opérateur sont conformes aux spécifications du langage source.
- En général, cette analyse est représentée par un arbre syntaxique décoré.

Exemple :

Il faut vérifier que la variable '**i**' possède bien le type '**entier'**, et que la variable '**a**' est bien un **nombre**. Cette opération s'effectue en parcourant l'arbre syntaxique et en vérifiant à chaque niveau que les opérations sont correctes.

2.5.4 Génération de code intermédiaire

- Après les phases d'analyse sémantique, certains compilateurs produisent une représentation intermédiaire, une sorte de code pour une machine abstraite.
- La forme intermédiaire "**code à trois adresses**" est largement utilisée.
- Le passage par le code intermédiaire apporte certains avantages :
    - Il est relativement facile de changer le code intermédiaire en code objet ;
    - La représentation intermédiaire permet d'appliquer des optimisations indépendantes du langage cible (code objet).

Exemple: **JAVA** utilise une forme intermédiaire spécifique: Les **bytecode.** Le bytecode est exécuté sur n'importe quelle plateforme à l'aide de la machine virtuelle JAVA (**JVM**)

2.5.5 Optimisation du code intermédiaire

- L'optimisation cherche à améliorer le code intermédiaire afin d'améliorer ses performances en termes de réduire le temps d'exécution ou l'occupation mémoire.
- Quelques-unes des opérations d'optimisation sont : Déplacement de code invariant, Elimination du code mort, Elimination des sous-expressions communes …etc.

- **Déplacement de code invariant :**

Le but est d'extraire du corps de la boucle des parties du code qui sont des invariants (nécessitent une seule exécution). Le nombre total d'instructions exécutées est diminué.

Exemple : Soit le code intermédiaire suivant :

```
for (int i = 0; i < n; i++) {
x = y + z;
a[i] = 6 * i + x * x; }
```

Après le déplacement du code invariant, on obtient :

```
x = y + z;
temp1 = x * x;
for (int i = 0; i < n; i++) {
a[i] = 6 * i + temp1; }
```

- **Elimination du code mort :**

Le compilateur sait reconnaitre les parties du code qui sont morts (la raison peut être une erreur de programmation)

Exemple : Des instructions qui sont inaccessibles à cause d'instructions de saut sont considérées comme du code mort et peuvent être éliminées :

```
if 1 <> 0 goto label
i := a[k]
k := k + 1
label:
```

- **Elimination des sous-expressions communes**

Dans le cas où la valeur d'une expression est calculée en plusieurs endroits du programme, l'optimiseur gère le résultat de cette expression et le réutilise plutôt que refaire le calcul.

Exemple : Soit le code intermédiaire suivant :

```
t6 = 4 * i
```

```
x = a[t6]
t7 = 4 * i
t8 = 4 * j
t9 = a[t8]
a[t7] = t9
t10 = 4 * j
a[t10] = x
```

Après l'élimination de sous-expressions communes (et quelques autres optimisations), on obtient :

```
t6 = 4 * i
x = a[t6]
t8 = 4 * j
t9 = a[t8]
a[t6] = t9
a[t8] = x
```

2.5.6 Génération de code objet

- Cette phase produit du code objet en :
  - Choisissant les emplacements mémoires pour les données ;
  - Sélectionnant le code machine pour implémenter les instructions du code intermédiaire ;
  - Allouant les registres

Exemple : voici le code intermédiaire optimisé suivant :

```
temp1 := 60.0 * id3
id1 := id2 + temp1
```

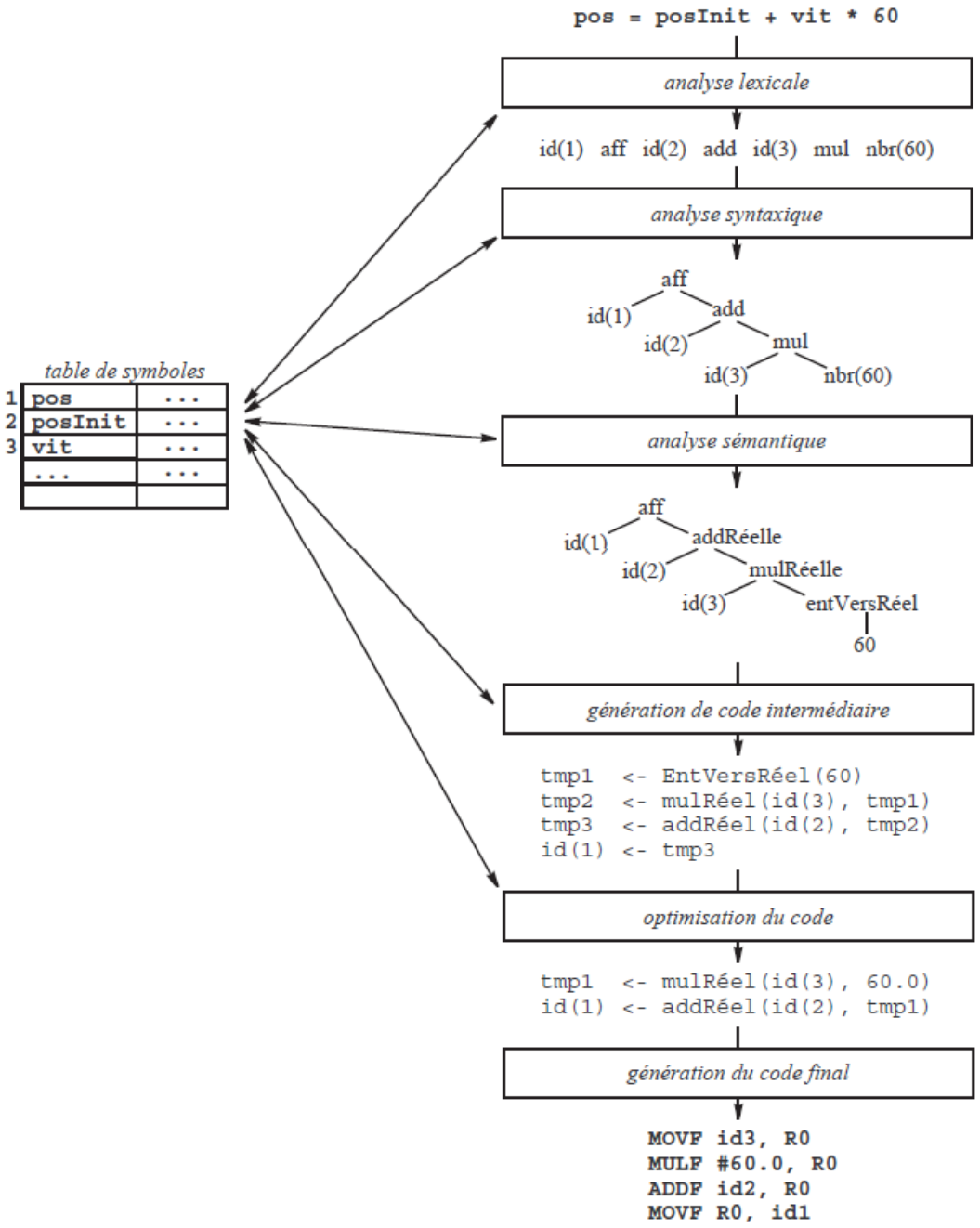
Voici le code objet généré en utilisant les registres R1 et R2:

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

2.5.7. Phases logiques de la compilation d'une instruction

La figure suivante illustre toutes les phases de compilation de l'instruction

pos := posInit + vit * 60


pos = posInit + vit * 60
analyse lexicale
id(1) aff id(2) add id(3) mul nbr(60)
analyse syntaxique
aff
id(1)
add
id(2)
mul
id(3)
nbr(60)
table de symboles
1 pos ...
2 posInit ...
3 vit ...
... ...
analyse sémantique
aff
id(1)
addRéelle
id(2)
mulRéelle
id(3)
entVersRéel
60
génération de code intermédiaire
tmp1 <- EntVersRéel(60)
tmp2 <- mulRéel(id(3), tmp1)
tmp3 <- addRéel(id(2), tmp2)
id(1) <- tmp3
optimisation du code
tmp1 <- mulRéel(id(3), 60.0)
id(1) <- addRéel(id(2), tmp1)
génération du code final
MOVF id3, R0
MULF #60.0, R0
ADDF id2, R0
MOVF R0, id1

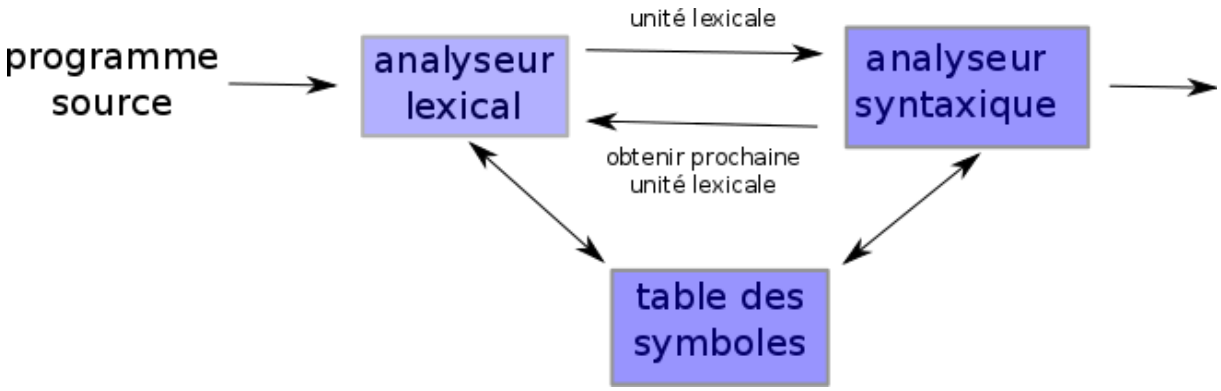
Chapitre 3. Analyse lexicale

3.1 Introduction

- L'analyse lexicale consiste à transformer d'un ensemble de caractères du programme source en **unités lexicales** afin de les fournir à **l'analyseur syntaxique**.
- Les unités lexicales les plus connues sont:
  – Mots clefs (for, if,. . .)
  – Symboles (+, , (,), . . .)
  – Identificateurs (toute suite de lettres et de chiffres qui commence par une lettre).
  – Nombres (toute suite de chiffres).
- Les commentaires et les blancs sont ignorés dans la phase d'analyse lexicale.

Dans l'analyse lexicale, on distingue les concepts suivants :

- **Unité lexicale** : correspond a une entité (un concept) renvoyé par l'analyseur lexical. Par exemple <; >;<=;>= sont tous des opérateurs relationnels.
- **Un lexème** : est une instance d'unité lexicale. Par exemple le lexème 6,28 une instance de l'unité lexicale nombre.
- **Un modèle** : associe des lexèmes a leur unité lexicale correspondante



3.2 Les langages formels

- On se donne un ensemble **Σ** appelé alphabet, dont les éléments sont appelés caractères.
- Un mot (sur **Σ**) est une séquence de caractères (de **Σ**). On note :
  - ε le mot vide,

- ***uv*** la concaténation des mots ***u*** et ***v*** (la concaténation est associative et **ε** est un élément neutre).
- Σ^* est l'ensemble des mots sur **Σ**.
- Σ^+ est l'ensemble des mots non vides sur **Σ**.

- Un langage sur **Σ** est un sous-ensemble L de Σ^*

Exemples*:*

- Σ_1 est l'alphabet et L_1 est l'ensemble des mots du dictionnaire français avec toutes leurs variations (pluriels, conjugaisons). L_2 est l'ensemble des phrases grammaticalement correctes de la langue française.
- Σ_2 est l'ensemble des caractères **ASCII**, et L_3 est composé de tous les mots clés de pseudo pascal : des symboles, des identificateurs, et de l'ensemble des entiers décimaux…. et L_4 est l'ensemble des programmes pseudo pascal.
- Σ_3 est {***a, b***} et L_5 est { $a^n b^n \mid n \in IN$} (tous les mots qui constituent de ***a*** et ***b*** et la langueur de ***a*** = la langueur de ***b***)

3.2.1 Type des langage formelles *:*

Les quatre types de langages formels sont classifiés selon la hiérarchie de Chomsky, qui organise les langages formels en fonction de leur complexité.

3.2.1.1 Langages de type 0 (langages récursivement énumérables):

Ce sont les langages les plus généraux, pouvant être reconnus par une machine de Turing. Il n'y a aucune restriction sur la forme des règles de production.

Exemple :

Le langage de toutes les chaînes de symboles, y compris des chaînes infinies ou incompréhensibles.

3.2.1.2 Langages de type 1 (langages contextuels):

Ces langages sont reconnus par une machine de Turing non-déterministe avec une mémoire limitée. Les règles de production sont de la forme $\alpha A\beta \rightarrow \alpha\gamma\beta$ où ***A*** est un symbole non-terminal, et ***γ*** n'est pas vide.

Exemple :

Le langage $L=\{a^n b^n c^m \mid n,m \geq 1\}$, où chaque chaîne contient un nombre égal de symboles ***a, b,*** et ***c***

3.2.1.3 Langages de type 2 (langages context-free ou sans contexte):

Ces langages sont générés par des grammaires hors contexte, où les règles de production sont de la forme $A \rightarrow \gamma$, avec A un non-terminal et γ une chaîne de terminaux et/ou de non-terminaux ou ε.

Exemple :

Le langage des parenthèses bien équilibrées, où chaque parenthèse ouvrante correspond à une parenthèse fermante

3.2.1.4 Langages de type 3 (langages réguliers):

Ce sont les langages les plus simples, pouvant être reconnus par un automate fini. Les règles de production sont de la forme $A \rightarrow aBA$ ou $A \rightarrow a$, où **A** et **B** sont des non-terminaux et a est un terminal.

Exemple :

Le langage $L=\{a^n b^n | n \geq 0\}$, qui contient des chaînes comme ***ab, aabb, aaabbb***, …etc

3.3 Expressions régulières :

Les expressions régulières représentent un formalisme simple permettant de décrire certains langages simples (les langages réguliers).

Elles permettent de décrire les unités lexicales d'une manière concise et compacte.

Le générateur d'analyseur lexical LEX utilise les expressions régulières pour spécifier ses unités lexicales.

Définition :

- On note ***a***, ***b***, et **c**. des lettres de l'alphabet **Σ**. ***M*** et ***N*** sont des expressions régulières et ***L[M]*** le langage associé à ***M***.
- Une lettre ***a*** désigne le langage {***a***}.
- Epsilon : **ε** désigne le langage {**ε**}.
- Concaténation : ***M N*** désigne le langage $L[M] \cap L[N]$.
- Alternative : ***M*** **|** ***N*** désigne le langage $L[M] \cup L[N]$.
- Répétition : M^* désigne le langage $(L[M])^*$.
- ***M*** **?** pour ***M*** **|** **ε**
- M^+ pour $M M^*$

Exemples :

- Lettre : [A-Za-z]
- Chiffre : [0-9]

- Identificateur : {lettre}({lettre}|{chiffre})*
  Ou bien [A-Za-z][A-Za-z0-9]*
- Nombre entier signé : [-+] ?{chiffre}+
  Ou bien[-+] ?[0-9]+
- Nombre réel : [-+] ?{chiffre}+(,{chiffre}+) ?
  Ou bien [-+] ? [0-9]+(,[0-9]+) ?

3.4 Automate à états finis :

Les langages sont reconnus par des machines formelles appelées : **automates** qui étant donnée un mot sont capable de dire s'il appartient ou pas à un langage.

Un langage sur un alphabet **Σ** est régulier si et seulement s'il est reconnu par un automate à états finis **[Théorème de Kleene]**. Alors, chaque expression régulière ***M***, possède un automate équivalent qui reconnaît ***L[M]***.

Un automate à états finis (**AF**) est un modèle d'un système et de son évolution, c'est-à-dire une description formelle du système et de la manière dont il se comporte.

Un automate à états finis (**AF**) est composé d'un ensemble fini d'états (représentés graphiquement par des cercles), d'un ensemble fini de transitions (représentés graphiquement par des arcs) et d'une fonction de transition décrivant l'action qui permet de passer d'un état à l'autre, d'un état initial, d'un ou plusieurs états finaux.

Un automate à états finis (**AF)** est donc un graph orienté où les nœuds correspondent aux états et les arcs contiennent les lettres de l'alphabet **Σ.**

Définition

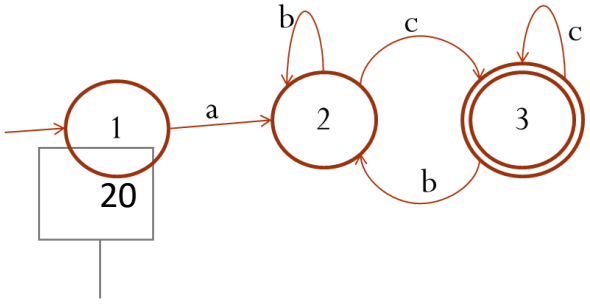
Un automate à états finis M est un multiple ($Q, \Sigma, \delta, q_0, F$) où :

- Σ : est un alphabet ;
- Q : est un ensemble fini d'états ;
- $\delta : Q \times \Sigma \rightarrow Q$ est la fonction de transition ;
- q_0 : est l'état initial ;
- F : est un ensemble d'états finaux.

Propriété : Le langage ***L(M)*** reconnu par l'automate ***M*** est l'ensemble **{w | δ(q₀ , w) ∈ F}** des mots permettant d'atteindre un état final à partir de l'état initial de l'automate.

Exemple :

Soit l'automate à états finis correspondant à l'expression régulière **ab*c(c*| b+c+)** :



Σ = {a, b,c}

Q = {1, 2, 3}

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

Etat initial = {*1}*

Etats finaux : un seul état final = {*3}*.

3.4.1 Représentation d'un automate :

- La fonction δ de domaine fini $Q \times \Sigma$ peut être représentée par une matrice de deux dimensions, dont les éléments sont :
  - Les états (pour un automate déterministe) ou
  - Un ensemble d'états (pour un automate non-déterministe)

Le tableau suivant représente la table de transition de l'automate précédent

	a	b	c
→{1}	{2}	-	-
{2}	-	{2}	{3}
#{3}	-	{2}	{3}

3.4.2. Automate à états finis déterministe (AFD) et non déterministe (AFN) :

- Un automate à états finis est **déterministe** (**AFD**) si : pour chaque lettre et chaque état on peut avoir qu'une seule transition sortante, et il ne possède pas de transitions par ε.
- Un automate à états finis est non déterministe **(AFN)** si : pour un état et une lettre on peut avoir plusieurs transitions sortantes, ou on peut avoir des transitions par ε.

3.4.3. Implémentation d'expression régulières :

Pour effectuer l'analyse lexicale sur les ordinateurs, les expressions régulières sont transformées en automates à états finis dont l'implémentation est simple et la reconnaissance des unités lexicales est rapide. Cette procédure passe par les 4 étapes suivantes :

- **1ère étape :** Transformation des expressions régulières en AFN.
- **2eme étape** : Transformation des AFN en AFD.
- **3eme étape** : Minimisation des AFD.
- **4eme étape** : Implémentation des AFD minimaux.

3.4.3.1. Transformation d'une expression régulière en AFN

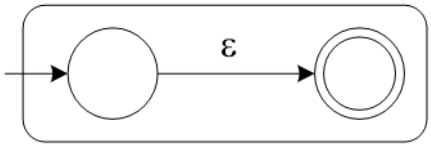
Construction de Thompson :

Parmi les méthodes les plus utilisées pour construire des automates à états finis à partir des expressions régulières, on trouve la **Construction de Thompson** qui génère automatiquement un **AFN** à partir d'une **ER** comme suit :

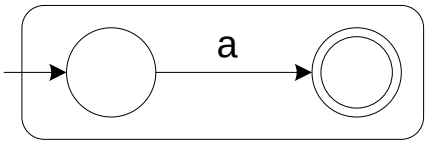
L'**ER** est décomposé en composants simples, et pour chaque composant on construit un automate selon les règles de Thompson de base. Ensuite les automates obtenus dans la première étape sont composés pour construire l'automate final selon les règles de composition de Thompson.

Règles de Thompson de base :

- **1ere règle :** permet de construire un automate pour l'expression régulière ε.

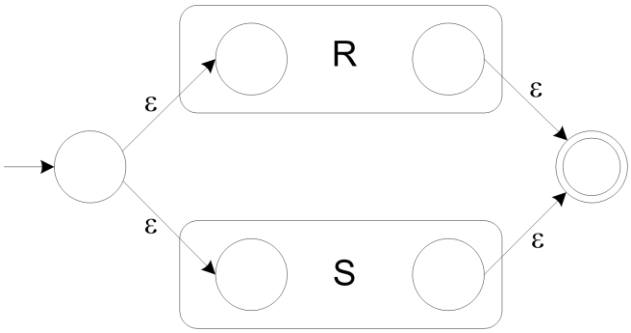


- **2eme règle :** permet de construire un automate pour l'expression régulière a.

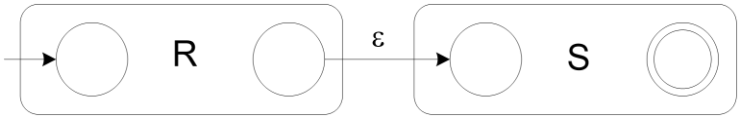


Règles de Thompson de composition

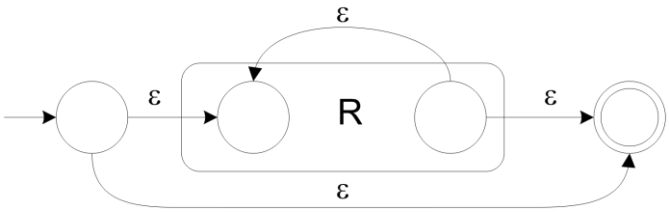
- **3eme règle : Alternation R|S :**



- **4eme règle : Concaténation RS :**

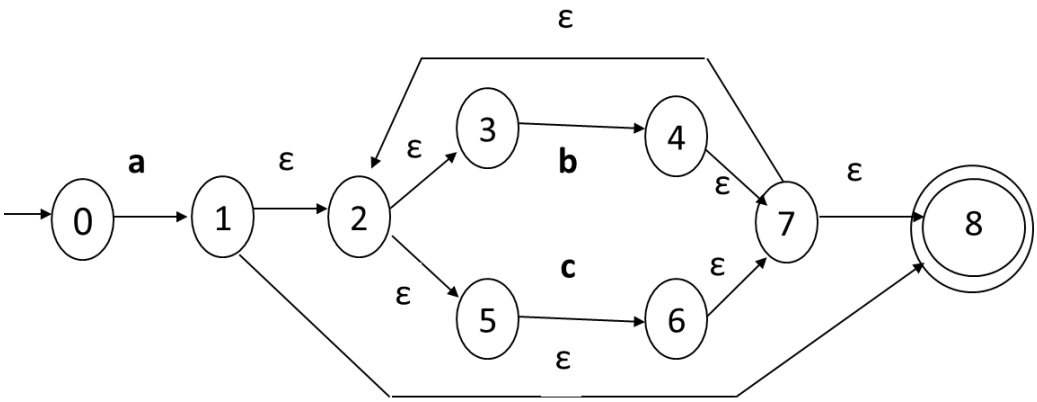


- **5eme règle : Etoile R* :**



Exemple

- **L'AFN obtenu à partir de l'ER: a(b|c)*.**



3.4.3.2 Transformation d'un AFN en AFD :

Algorithme de transformation

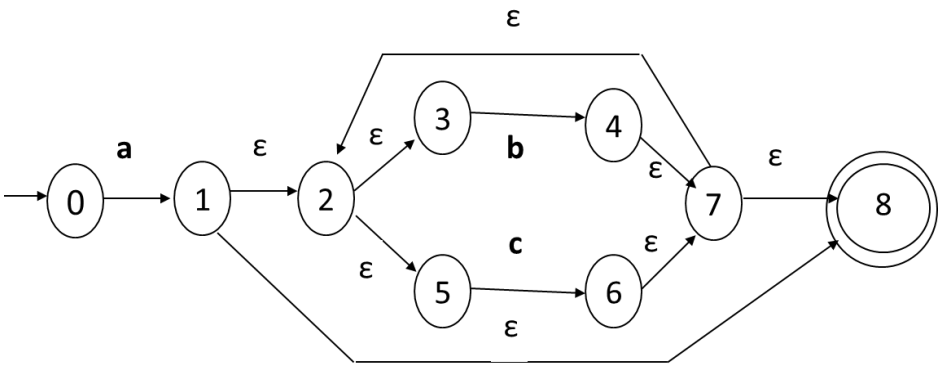
- **Données :** Un **AFN** définissant le langage que **N.**
- **Résultat :** Un **AFD** définissant le même langage que **N**.
- **D** est la table de transition de **AFD.**

  On dispose des fonctions suivantes :
- **ε-fermeture(e) :** ensemble des états de l'AFN accessibles depuis l'état e de l'AFN par ε-transitions (état e inclus).
- **ε-fermeture(T) :** ensemble des états de l'AFN accessibles depuis un état e appartenant à T par des ε-transitions (l'ensemble T inclus).
- **Transiter(T, a) :** ensemble des états de l'AFN vers lesquels il existe une transition dans l'AFN sur le symbole a à partir d'un état e appartenant à T**.**

```
E0 = ε-fermeture(Etat initial(AFN));
ajouter E0 comme état initial de D (sans le marquer);
 tant que un état E de D est non marque faire
     marquer E;
      pour chaque caractère c de l'alphabet faire
            F = ε-fermeture (transiter(E, c));
            si F n'est pas un état de D alors
                  ajouter l'état F a D (sans le marquer);
                  si un élément de F est un état d'acceptation de AFN alors
                        F est un état d'acceptation de D
                  fin si
            fin si
           ajouter la transition E --c--> F  à  D;
      fin pour
 fin tant que
fin
```

Exemple

Prenons l'exemple de AFN de la Figure II.2 correspondant à l'expression régulière **a.(b|c)***



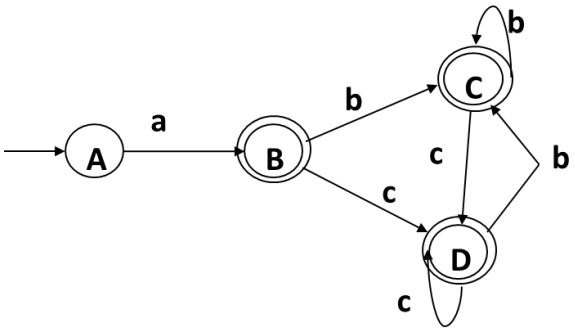
Construction de l'AFD :

- **ε-fermeture(0) = {0}**
- **Table de transition D de l'AFD :**

	a	b	c
→ **A ={0}**	{1,2,3,5,8}=**B**	-	-
B#={1,2,3,,5,8}	-	{4,7,8,2,3,5}=**C**	{6,7,8,2,3,5}=**D**
C#={4,7,8,2,3,5}	-	**C**	**D**
D#={6,7,8,2,3,5}		**C**	**D**

- **Etat initial :** ε-fermeture(0) = {0}=**A**
- **Etats finaux : B, C , D**

La figure suivant représente l'AFD obtenu pour l'ER : a.(b|c)* en utilisant l'algorithme de transformation.



3.4.3.3. Minimisation de l'AFD :

Pour effectuer l'analyse lexicale, il est préférable que la table d'analyse de L'AFD soit la plus petite possible afin d'économiser la mémoire.

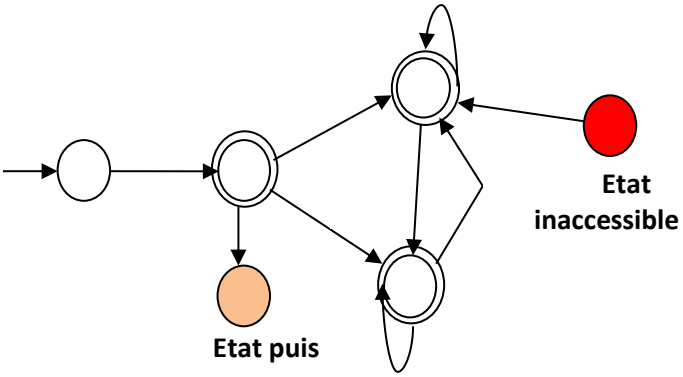
Théorème : Il existe un automate déterministe unique avec un nombre minimal d'états reconnaissant un langage rationnel ***L*** **[Myhill-Nerode].**

L'algorithmes de raffinements de partitions (ex: l'algorithme de **Moore** et l'algorithme de **Hopcroft**) sont les plus simple à utiliser.

Remarque :

Avant l'utilisation de l'algorithmes de raffinements de partitions. Il faut supprimer les états **inaccessibles** et les états **puis**.

- **Les états inaccessibles** sont des états ou on ne peut pas atteindre cet état à partir d'un état initial.
- **Les états puis** sont les états ou il n'a pas un chemin vers un état final à partir de cet état.

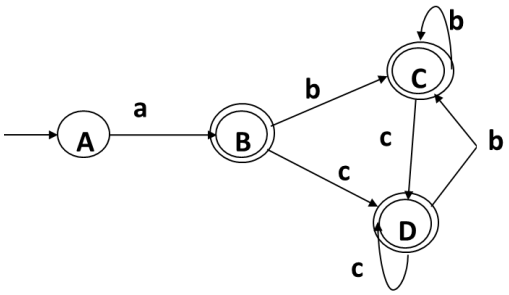


Algorithme de minimisation :

Soit A= $(Q, \Sigma, \delta, q_0, F)$ un automate fini déterministe
- Initialement: Créer deux Classes d'états C1 et C2 // C1 contient les états finaux (F) et C2 contient les états non-finaux (Q\F)

Répéter
  Pour chaque partition Ci **faire**
    Pour chaque symbole d'entré a **faire**
      Si il existe deux états différents q1 et q2 appartient à Ci qui lisent le symbole a et mènent vers des états appartient aux deux classes différentes **alors**
        Créer une nouvelle classe Cj et séparer q1 de q2.
      Fin si
    Fin
  Fin pour
Jusqu'à ce que il n y a pas de deux classes à séparer.

Exemple : Prenons l'exemple de AFN correspondant à l'expression régulière **a.(b|c)***



- **Initialement:**

I: C1: {A} , C2:{ B,C,D}

B---b--> **C** **C**---b--> **C** **D**---c--> **D**

B---c--> **D** **C**---c--> **D** **D**---c--> **D**

- **1ière Itération:**

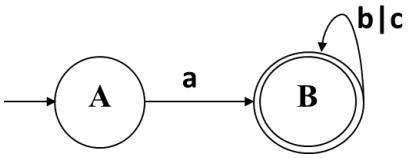
II : C1: {A} , C2:{ B,C,D}

On ne peut scinder **{B,C,D}** puisque **B, C et D** sont des états inséparables.

Alors on obtient la Table de transition de l'AFD minimal suivante :

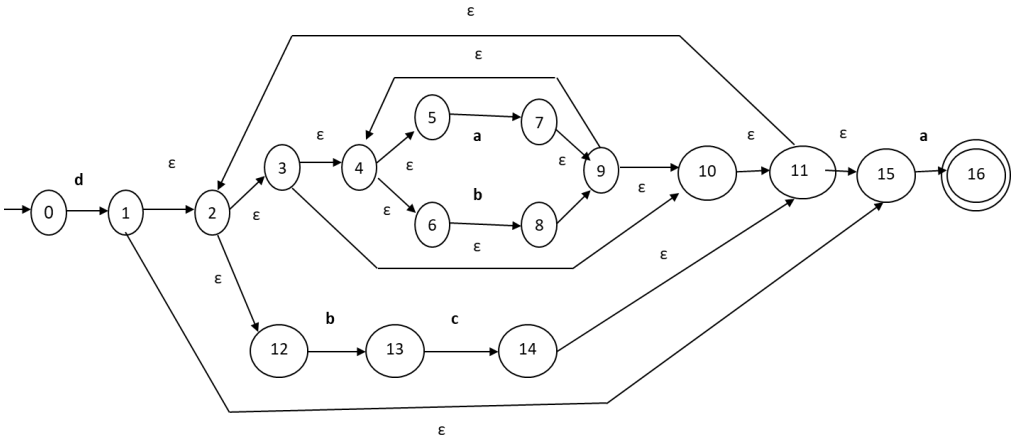
	a	b	c
→A	B		-
B#	-	B	B

Et on obtient le AFD minimal pour l'ER: **a.(b|c)*** suivant :



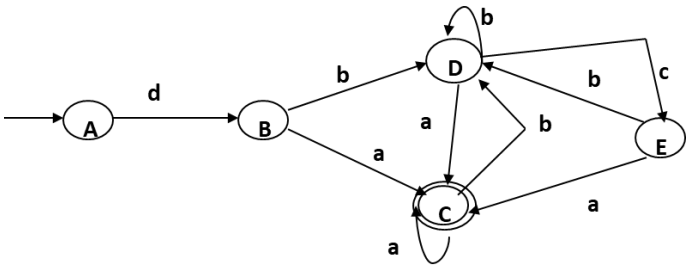
Exemple 2 :

- **L'AFN obtenu a partir de l'ER: d((a|b)*|bc)*a.**
- **ER →AFN**



- **AFN →AFD**

	a	b	c	d
→A = {0}	-	-	-	{1,2,3,4,5,6,10,11 ,12,15}=**B**
B	{7,9,10,11,15,4,5,6,2 ,3,12,16}=**C**	{8,13,9,10,11,15,4, 5,6,2,3,12}=**D**	-	-
C#	C	D	-	-
D	C	D	{14,11,15,2,3,4,5, 6,10,12}=**E**	-
E	C	D	-	-



- **AFD →AFD Minimal**
- **Initialement:** I: {A,B,E,D} , {C}
- **1ière Itération:** II : {A},{B,E},{D}, {C}

$A \xrightarrow{a} \Phi$

$B \xrightarrow{a} C$ $B \xrightarrow{b} D$ $B \xrightarrow{c} \Phi$ $B \xrightarrow{d} \Phi$

$E \xrightarrow{a} C$ $E \xrightarrow{b} D$ $E \xrightarrow{c} \Phi$ $E \xrightarrow{d} \Phi$

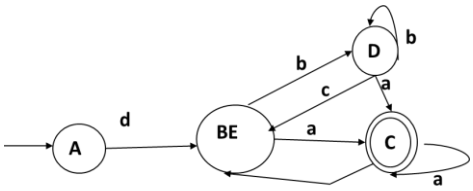
$D \xrightarrow{a} C$ $D \xrightarrow{b} D$ $D \xrightarrow{c} E$

- **2ième Itération**: III: {A},{D},{B,E}, {C}
- II = III. On ne peut scinder {B,E} puisque B et E mènent vers les mêmes états avec tous les symboles d'entrés.

On obtient la table d'analyse suivante pour l'AFD minimal

	a	b	c	d
→A	-	-	-	BE
BE	C	D		
D	C	D	BE	-
C#	C	D	-	-

On obtient l'automate AFD minimal suivant



3.4.3.4. Implémentation des AFD minimaux :

Algorithme de reconnaissance

- **Entrée :** une chaine de caractère d'entrée **S** terminée par un caractère spécial "#".
- **Sortie :** Chaine reconnue par l'automate ou non.
- On dispose des fonctions suivantes :

✓ **Transiter(e, c) :** donne l'état de l'automate vers lequel il existe une transition depuis l'état e sur le caractère d'entrée **c.**

✓ **CarSuiv ( ) :** retourne le prochain caractère à analyser de la chaine **S.**

```
e:=e0;
c:=CarSuiv();
Tant que (c ≠ '#' et e ≠ ∅)
Faire
        e:=Transiter(e,c);
        c:=CarSuiv();
Fait;
Si e∈F
Alors "Chaine acceptée"
Sinon "Chaine refusée"
Fsi
```

3.4.4 Outils pour Implémenter les expressions régulières (LEX)

Les automates d'états finis jouent un rôle clé dans l'analyse lexicale. Ils peuvent être implémentés en utilisant des langages comme Java, C++, ou PHP, ou bien via des générateurs d'analyseurs lexicaux tels que Lex, Flex, et JLex. Ces outils permettent de simplifier le processus de détection de unités lexicales (tokens) et d'analyse des chaînes de caractères, étapes essentielles dans la construction des compilateurs. Dans la suite nous allons introduire le générateur d'analyseurs lexicaux **LEX**.

3.4.4.1. Définition

Lex est un outil de génération d'analyseurs lexicaux en langage **C**. Il a été originellement écrit par **Mike Lesk** et **Eric Schmidt** en 1975. IL est capable de traiter des langages de type 3 (réguliers), et il est fréquemment utilisé en association avec le générateur d'analyseur syntaxique **Yacc.**

Le principe de Lex consiste à prendre en entrée la définition des unités lexicales sous forme d'expressions régulières et à produire un automate fini déterministe minimal capable de reconnaître ces unités lexicales. Lex génère cet automate sous la forme d'un programme en langage **C**.

3.4.4.2. Expressions régulières en LEX

Le tableau suivant présente les principales unités lexicales des LEX.

Symbole	Signification
x	Le caractère '**x**'
.	N'importe quel caractère sauf **\n**
[xyz]	Soit **x**, soit **y**, soit **z**
[^bz]	Tous les caractères, sauf **b** et **z**
[a-z]	N'importe quel caractère entre **a** et **z**
[^a-z]	Tous les caractères, sauf ceux compris entre **a** et **z**
R*	Zéro **R** ou plus, ou **R** est n'importe quelle expression régulière
R+	Un **R** ou plus
R?	Zéro ou un **R** (c'est-a-dire un **R** optionnel)
R{2,5}	Entre deux et cinq **R**
R{2,}	Deux **R** ou plus
R{2}	Exactement deux **R**
"[xyz\"foo	La chaine '**[xyz"foo**'
{NOTION}	L'expansion de la notion **NOTION** definie plus haut
RS	**R** suivi de **S**
R|S	**R** ou **S**
R/S	**R**, seulement s'il est suivi par **S**
^R	**R**, mais seulement en début de ligne
R$	**R**, mais seulement en fin de ligne
<<EOF>>	Fin de fichier

Exemples

- **blancs [\t\n]+**
- **lettre [A-Za-z]**
- **chiffre10 [0-9] /* base 10 */**
- **chiffre16 [0-9A-Fa-f] /* base 16 */**
- **identificateur {lettre}(_|{lettre}|{chiffre10})***

  ou bien identificateur [A-Za-z] [_A-Za-z0-9] *
- **chiffre [0-9]**
- **entier {chiffre}+**

- **exposant [eE][+-]?{entier}**
- **reelVF {entier}("."{entier})?{exposant}?**
- **reel [+-]? [0-9] +("." [0-9] +)?**

3.4.4.3. Structure d'un programme LEX :

Un fichier de description pour Lex est formé de trois parties :

- Déclarations
- %%
- Productions
- %%
- Code additionnel

Aucune partie n'est obligatoire.

Le symbole %% est utiliser comme un séparateur entre les parties

- **1ere partie** : **Déclarations :** peut contenir :

Du code écrit dans le langage cible (**C**), encadré par **%{** et **%}**. **Lex** recopie tel quel tout ce qui est écrit entre ces deux signes.

Des expressions régulières définissant des notions non terminales, à utiliser dans le reste de la première partie du fichier **lex**, ainsi que dans la seconde partie, en les mettant entre **{**et**}**. Ces spécifications sont de la forme :

Notion expression régulière

Exemple :

%{

#include "calc.h"

#include <stdio.h>

#include <stdlib.h>

%}

/* Expressions régulières */

Blancs [\t\n]+

Lettre [A-Za-z]

chiffre [0-9]

identificateur {lettre}(_|{lettre}|{chiffre10})*

- **2eme partie** : **les productions :**

Cette partie sert à indiquer à **Lex** ce qu'il devra faire lorsqu'il rencontrera telle ou telle **unité lexicale**. Elle peut contenir des **productions** de la forme :

Expression régulière action

Les **actions** seront écrites dans le code du langage cible **(C)** et doivent être placées entre **{**et**}**.

Si **action** est **absente**, **Lex** recopiera les caractères tels quels sur la sortie standard.

Les commentaires tels que **/* ... */** peuvent être placés **uniquement** dans les **actions** entre parenthèses. Dans le cas contraire, ceux-ci seraient considérés par **Lex** comme **des expressions régulières** ou **des actions**, ce qui donnerait lieu à des **messages d'erreur**.

La variable **yytext** désigne dans les **actions** les caractères acceptés par une **expression régulière**. Il s'agit d'un tableau de caractère de longueur **yyleng** (donc défini comme **char yytext[yyleng]**).

Exemple :

```
%%
[ \t]+$      ;
[ \t]        printf(" ");
```

Ce programme supprime tous les espaces inutiles dans un fichier.

- **3eme partie**: **Le code additionnel:**

On peut mettre dans cette partie **facultative** tous le code qu'on veut. Si on ne met rien, **Lex** considère que c'est juste :

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

3.4.4.4. Exécution de LEX sous Linux :

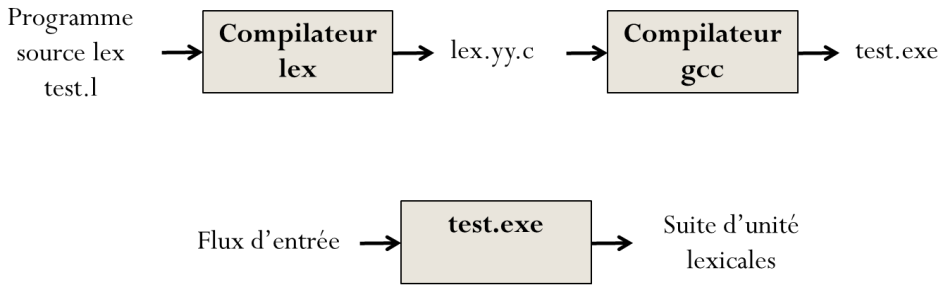
Le programme LEX s'exécute comme suit :

- Le fichier nommé par exemple **test.l** qui contient la spécification de l'analyseur lexical à produire est compiler avec le compilateur **lex** en exécutant la commande **lex**.
- La commande **lex** génère le code **C** de l'analyseur qui est mis dans un fichier **lex.yy.c**
- Le compilateur **gcc** est utilisé pour compiler le fichier **lex.yy.c** et générer un code exécutable (**a.out** par exemple).
- Le code exécutable est chargé et exécuté

lex test.l

gcc lex.yy.c -o test.exe -lfl

./test.exe



3.4.4.5. Exemple d'un programme lex :

Le programme en **lex** en dessous est conçu pour compter le nombre des caractères, mots, et lignes dans un fichier.

```
/* just like Unix wc */
%{
int chars = 0;
int words = 0;
int lines = 0;
%}
%%
[a-zA-Z]+   { words++; chars += yyleng; }
\n          { chars++; lines++; }
.           { chars++; }
%%
main(int argc, char **argv)
{
 yyin = fopen("lex.yy.c", "r");
 yylex();
 printf("lines=%d\n words=%d\n words=%d", lines, words, chars);
  fclose(yyin);
}
```

Chapitre 4. Analyse syntaxique

4.1 Introduction :

L'analyse syntaxique à plusieurs objectifs :

- Déterminer si la suite de tokens (unité lexicales) est conforme à la grammaire définissant le langage source ;
- Repérer les erreurs de nature syntaxique ;
- Définir une structure hiérarchique (un arbre syntaxique).

Les grammaires **algébriques** ou **non contextuelles** sont suffisamment puissantes pour décrire la partie principale de la syntaxe de la plupart des langages de programmation.

Dans ce chapitre nous allons faire quelques rappels sur les grammaires non-contextuelles.

4.2 Grammaire algébrique (non contextuelle) :

Une **grammaire algébrique**, ou **grammaire non contextuelle**, aussi appelée grammaire **hors-contexte** ou grammaire « **context-free** » est une grammaire formelle dans laquelle chaque règle de production est de la forme :

$$\mathbf{A \rightarrow \alpha}$$

Où **A** est un symbole non terminal et α est une chaîne composée de **terminaux** et/ou de **non-terminaux**.

Le terme « **non contextuel** » provient du fait qu'un non terminal **A** peut être remplacé par α, sans tenir compte du contexte où il apparaît.

Une grammaire non-contextuelle G est un quadruplet **<N,T,P,S>** où :

- **N** : ensemble fini non vide de symboles appelés symboles **non terminaux**,
- **T** : ensemble fini de symboles appelés symboles **terminaux**,
- **S** : $S \in N$, **axiome** de la grammaire,
- **P** : ensemble de **productions**,

  Chaque production est de la forme:

$$\mathbf{A \rightarrow \alpha \quad où}$$

$$\mathbf{A \in N \text{ et } \alpha \in (N \cup T)^*}$$

Exemple

La grammaire G décrivant les expressions arithmétiques :

G = <{S}, {+,-,*,/,(,),a,b,c},P,S>

P :

S → S + S

S → S – S

S → S * S

S → S / S

S → (S)

S → a

S → b

S → c

Autre manière :

P : S → S + S | S – S | S * S | S / S | (S) | a | b | c

4.3 Dérivations et arbres de dérivation :

Il existe deux façons pour décrire l'appartenance d'une **phrase** au **langage générée** à partir d'une grammaire donnée :

- La première consiste à lister une suite des applications des règles (**suite de dérivation**).
- La deuxième synthétise cette liste en un arbre, appelé (**arbre de dérivation**).

4.3.1 Dérivations :

La dérivation est le processus par lequel une grammaire définit un langage s'appelle ***dérivation*** :

- Soit $G = (N, T, P, S)$ une grammaire non contextuelle,
- $A \in N$ un symbole non terminal et $\gamma \in (N \cup T)^*$ une suite de symboles, tels qu'il existe dans P une production $A \rightarrow \gamma$.
- Quelles que soient les suites de symboles α et β, on dit que $\alpha A \beta$ ***se dérive en une étape*** en la suite $\alpha\gamma\beta$ ce qui s'écrit :

$$\alpha A \beta \Rightarrow \alpha\gamma\beta$$

Suite de dérivations :

- Si $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n$ on dit que α_0 se dérive en α_n en n étapes, et on écrit :

$$\alpha_0 \Rightarrow^n \alpha_n.$$

- Si α se dérive en β en un nombre quelconque, éventuellement nul, d'étapes on dit simplement que α se dérive en β et on écrit:

$$\alpha \Rightarrow^* \beta.$$

- Si α se dérive en β en un nombre quelconque, non nul, d'étapes on dit simplement que α se dérive en β et on écrit:

$$\alpha \Rightarrow^+ \beta.$$

Exemple :

Soit la grammaire G_1 :

EXPRESSION → EXPRESSION "+" TERME | TERME

TERME → TERME "*" FACTEUR | FACTEUR

FACTEUR → nombre | identificateur | "(" EXPRESSION ")"

- La chaîne **w = nombre " * " identificateur " + " nombre** dérive de **EXPRESSION** de la manière suivante:
- **EXPRESSION ⇒ EXPRESSION "+" TERME ⇒ TERME "+" TERME ⇒ TERME "*" FACTEUR "+" TERME ⇒ FACTEUR "*" FACTEUR "+" TERME ⇒ nombre "*" FACTEUR "+" TERME ⇒ nombre "*" identificateur "+" TERME ⇒ nombre "*" identificateur "+" FACTEUR ⇒ nombre "*" identificateur "+" nombre**

4.3.1.1 Dérivation la plus à gauche et Dérivation la plus à droite :

Il existe deux types de dérivations ; la dérivation la plus à gauche et la dérivation la plus à droite.

La dérivation la plus à gauche est entièrement composée de dérivations en une étape dans lesquelles à chaque fois c'est le non-terminal le plus à gauche qui est réécrit.

La dérivation la plus à droite est entièrement composée de dérivations en une étape où, à chaque étape, c'est le non-terminal le plus à droite qui est réécrit.

Exemple :

- Soit la grammaire suivante **:**

  G = <{S,A}, {a,b},P,S>

  P : S → aAS | a

  A → SbA | SS | ba

- La chaine **aabbaa** dérive de l'axiome **S** en utilisant **:**

- **La dérivation la plus à gauche :**

  S ⇒ aAS ⇒ aSbAS ⇒ aabAS ⇒ aabbaS ⇒ aabbaa

- **La dérivation la plus à droite :**

  S ⇒ aAS ⇒ aAa ⇒ aSbAa ⇒ aSbbaa ⇒ aabbaa

4.3.1.2 Langage engendré par une grammaire :

- **Soit** $G = (N, T, P, S)$ une grammaire non contextuelle ; le langage engendré par *G* est l'ensemble des chaînes de symboles terminaux qui dérivent de ***S*:**

$$L(G) = \{ w \in T^* \mid S \Rightarrow^* w\}$$

  - Si une chaîne $v \in L(G)$ on dit que ***v*** est une ***phrase*** **de *G*.**
  - Plus généralement, si $\alpha \in (T \cup N)^*$ est tel que $S \Rightarrow^* \alpha$ alors on dit que α est une ***proto-phrase*** **de *G*.**

Une **proto-phrase** dont tous les symboles sont terminaux est une **phrase.**

Exemple1 :

- Soit la grammaire **G1=<N,T,P,S>** avec **:**

N = {S} , T = {a,b} ,

P = {S→ aSb ; S→ε }

L(G) = {$a^n b^n$ | n ≥ 0}

Exemple2 :

Soit la grammaire **G2=<N,T,P,S>** avec **:**

N = { S } ,T = { a, b }

P = { S → aaS | bbS | ε }

L(G) = {(aa | bb)*}

4.3.2 Arbres de dérivation :

Soit ***w*** une chaîne de symboles terminaux du langage ***L*(*G*)** ; il existe donc une dérivation telle que $S \Rightarrow^* w$. Cette dérivation peut être représentée graphiquement par un arbre, appelé **arbre de dérivation**, défini de la manière suivante :

- La racine de l'arbre est le symbole de départ **S**;
- Les nœuds intérieurs sont étiquetés par des symboles non terminaux ;
- Si un nœud intérieur ***e*** est étiqueté par le symbole ***A*** et $A \rightarrow A_1 A_2 \ldots A_k$ est une production de la grammaire alors les fils de ***e*** sont des nœuds étiquetés, de la gauche vers la droite, par $A_1, A_2 \ldots A_k$ **;**
- Les feuilles sont étiquetées par des symboles terminaux.

Exemple :

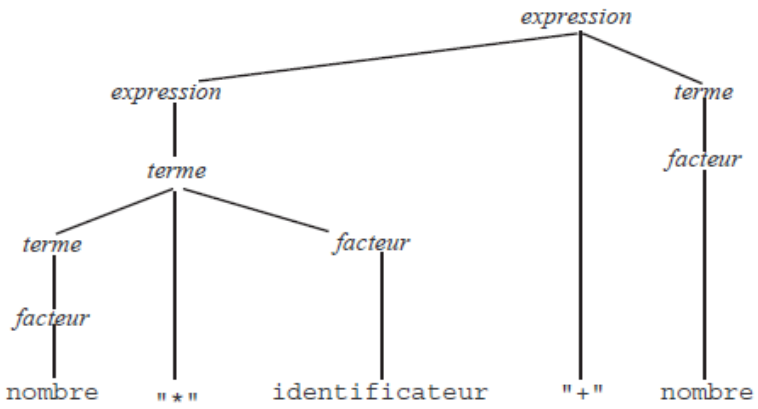
Soit la grammaire dont ses productions sont les suivantes :

EXPRESSION → EXPRESSION "+" TERME | TERME

TERME → TERME "*" FACTEUR | FACTEUR

FACTEUR → nombre | identificateur | "(" EXPRESSION ")"

L'arbre de dérivation pour la chaîne : **nombre " * " identificateur " + " nombre**



4.3.2.1 Grammaire ambiguë :

Une grammaire est **ambiguë** s'il existe **plusieurs dérivations gauches (ou droite) différentes** pour une même chaîne de terminaux.

Une grammaire est **ambiguë** lorsqu'au moins un mot engendré par la grammaire possède au moins **deux arbres de dérivation distincts**.

On notera que l'ordre des dérivations (gauche, droite) ne se voit pas sur l'arbre de dérivation

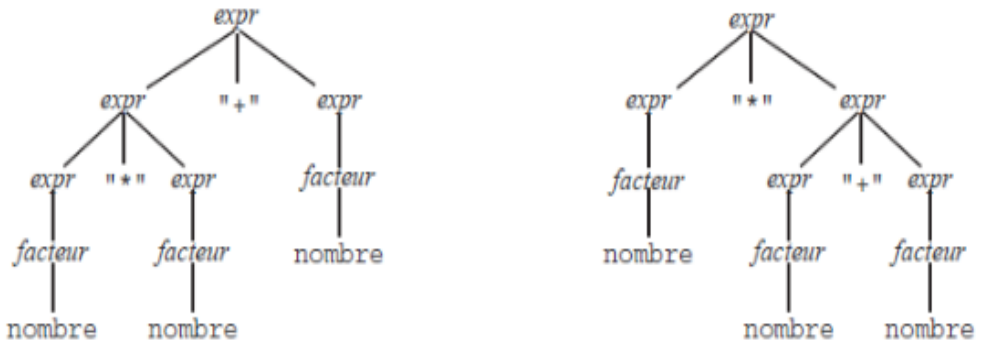
Exemple :

- Soit la grammaire **suivante** :

  exp* → *exp* "+" *exp* | *exp* "*" *exp* | *facteur

  ***facteur* → nombre | ident | "(" *exp* ")"**

- Cette grammaire est **ambiguë** car elle possède **deux arbres de dérivation distincts pour la chaîne: nombre "*" nombre "+" nombre**



4.4 Analyses syntaxique descendantes et ascendantes :

La plupart des méthodes d'analyse syntaxique tombent dans **deux classes** :

- **Les méthodes d'analyse descendantes** (**top-down parsing en anglais**). Elles sont les plus populaires car les analyses correspondantes sont **faciles** à mettre en œuvre à la main ;
- **Les méthodes d'analyse ascendantes** (**bottom-up parsing en anglais**). Elles sont les plus **puissantes** et sont celles utilisées par les outils qui génèrent automatiquement un analyseur à partir d'une grammaire (exemple : **yacc**).

4.4.1 Analyse descendante :

Dans les analyses **descendantes** : l'arbre d'analyse est construit en partant de la **racine** et en allant jusqu'au **feuilles** ;

L'analyse s'appuie sur une **dérivation à gauche** de la chaîne, c'est-à-dire qu'on remplace toujours en premier le non-terminal le **plus à gauche.**

Exemple : Soit la grammaire **G=<N,T,P,S>** avec :

N = {**E,T,F**} , T = {**+,*,),(,i**} ,

P = {

E → E + T | T

T → T × F | F

F → (E) | i

}

On présente ci-dessous la dérivation de la chaîne **(i + i) * i** depuis la racine de l'arbre d'analyse.

E	(Étape 0)
E ⇒ T	(Étape 1)
⇒ T * F	(Étape 2)
⇒ (E) * F	(Étape 3)
⇒ (E+T) * F	(Étape 4)
⇒ (T+T) * F	(Étape 5)
⇒ (F+T) * F	(Étape 6)
⇒ (i+T) * F	(Étape 7)
⇒ (i+F) * F	(Étape 8)
⇒ (i+i) * F	(Étape 9)
⇒ **(i+i) * i**	(Étape 10)

4.4.2 Analyse ascendante :

Dans les analyses ascendantes on agit **inversement :**

L'arbre est construit **en remontant** à la racine depuis les feuilles en utilisant des **dérivations inverses ;**

L'analyse s'appuie sur une dérivation à droite de la chaîne c'est-à-dire qu'on remplace toujours en premier le non-terminal le plus à droite.

Exemple :

Soit la grammaire **G=<N,T,P,S>** avec :

N = {**E,T,F**} , T = {**+,*,),(,i**} ,

P = {

E → E + T | T

T → T × F | F

F → (E) | i

}

On présente ci-dessous la dérivation inverse de la chaîne **(i + i) * i** depuis les feuilles de l'arbre d'analyse.

(i+i)*i (Étape 0)

⇐(F+i)*i (Étape 1)

⇐(T+i)*i (Étape 2)

⇐(E+i)*i (Étape 3)

⇐(E+F)*i (Étape 4)

⇐(E+T)*i (Étape 5)

⇐(E)*i (Étape 6)

⇐F*i (Étape 7)

⇐T*i (Étape 8)

⇐T*F (Étape 9)

⇐T (Étape 10)

⇐**E** (Étape 11)

4.5 Traitement des erreurs syntaxiques :

En général, **plus grande** part du traitement des **erreurs** est faite par le **parser (analyseur syntaxique)** car une grande partie des **erreurs** effectuées sont par nature **syntaxiques.**

Un **gestionnaire des erreurs** dans un **parser** doit:

- Indiquer la présence d'erreurs de façon claire et précise le **lieu** et la **cause** (**le lieu** : **exemple** : visualiser la **ligne** erronée, avec marqueur désignant **la position de l'erreur. La cause** : - **parenthèse fermante** en plus - **point-virgule** manquant)
- Traiter chaque erreur rapidement.
- Ne pas ralentir de façon significative la compilation d'un programme correct.

Il existe plusieurs techniques de récupération sur erreur. En particulier :

- **Mode panique,**
- **Correction locale**
- **Productions d'erreurs**
- **Correction globale**

4.5.1 Récupération sur erreur en mode panique :

Cette méthode est la plus simple à **implanter**, applicable dans la **plupart** des méthodes d'analyse.

Pendant le déroulement de l'analyse syntaxique, un ensemble d'unités lexicales (**symboles**) **de synchronisation** est continuellement mis à jour. Exemples (le symbole **End if** dans une structure conditionnelle **if**. **Le point-virgule** dans une **instruction d'affectation**).

Lorsqu'une erreur est détectée, on saute (ignore) tous les symboles donnés par le **scanner** jusqu'à ce qu'on rencontre un **symbole de synchronisation.**

- **Avantages**
  - Simple
  - Ne boucle pas à l'infini
- **Inconvénients**
  - Une partie considérable du programme **peut être sautée** sans vérifier sa validité
- **Conclusion**
  - **La récupération en mode panique** est très adéquate dans le cas où les erreurs multiples sont rares dans une même instruction

4.5.2 Récupération sur erreur en mode correction locale :

Lorsqu'une erreur est détectée, des **corrections locales** sont effectués en **modifiant** le **préfixe** du texte source restant de l'instruction en question. (**Exemples** : -remplacement d'une virgule par un point-virgule. -Suppression d'un point-virgule. - insertion d'un point-virgule manquant)

Le **remplacement** choisi ne doit pas conduire à **une boucle infinie** (**Exemple** : si on insère toujours quelque chose devant le symbole courant de la chaîne)

- **Inconvénient** :

Difficulté à gérer les situations dans lesquelles l'erreur réelle s'est produite avant le point de détection.

4.5.3 Récupération sur erreur en mode production d'erreurs

L'utilisation **de règles de production d'erreurs** est une méthode de **récupération sur erreur**.

Elle consiste à ajouter à la grammaire d'un langage **des productions contenant la notion d'erreur**.

Cette technique est notamment utilisée par **Yacc**.

Exemple :

Soit la grammaire **G=<N,T,P,S>** avec :

N = {*Expr*, *Op*} , T = {+,-,*,/,id} , **P = {** :

***Expr* → *Expr Op Expr* |(*Expr*) |- *Expr* | id**

***Op* → + | - | * | / }**

Pour détecter une parenthèse fermante mal placée, il suffit d'ajouter les règles :

***Erreur → Op*) | *Expr*)**

- **Inconvénient** : Il faut un grand nombre de règles pour traiter un grand nombre d'erreurs.

4.5.4 Récupération sur erreur en mode correction globale

Lorsqu'une chaîne d'entrée erronée (**instruction X**) , l'analyseur crée un arbre d'analyse pour une **instruction Y** sans erreur la plus proche de **X**. Cela peut permettre à l'analyseur d'apporter des modifications minimes dans le code source en remplaçant **X** par **Y.**

- **Inconvénient :**

Méthode trop coûteuse en mémoire et en temps. Pour l'instant, elle n'a pas encore été mise en œuvre dans la pratique.

4.6 Yacc, un générateur d'analyseurs syntaxiques :

Le programme **Yacc** (**Y**et **A**nother **C**ompiler **C**ompiler) est un **générateur d'analyseurs syntaxiques**.

Il prend en entrée un **fichier source constitué** essentiellement **des productions** d'une grammaire non contextuelle **G**, et il donne comme sortie : un programme **C** qui, une fois compilé, il donne **un analyseur syntaxique** pour le **langage L(G)**.

Yacc s'appuie sur la méthode d'analyse syntaxique **ascendante LALR**.

Chapitre 4. Analyse syntaxique

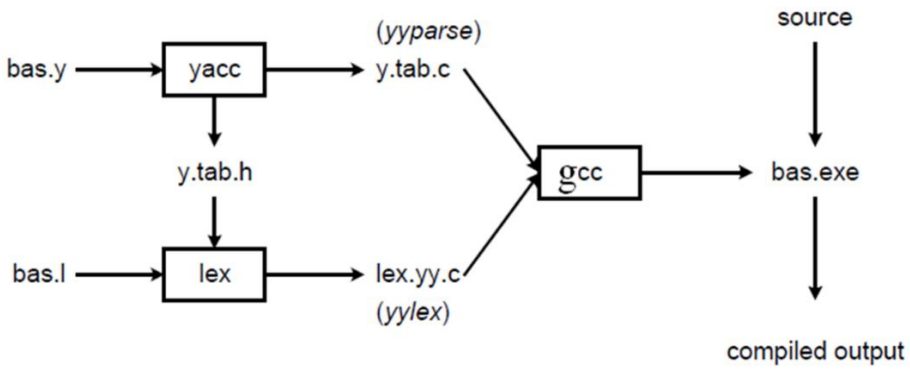
Dans la description de la grammaire donnée à **yacc** on peut associer **des actions sémantiques** aux **productions** ; ce sont des instructions de code source **C** que **yacc** place aux bons endroits de **l'analyseur construit**. Ce dernier peut ainsi exécuter des **actions** ou produire **des informations déduites** du texte source.

4.6.1 Yacc et lex :

Un analyseur syntaxique requiert pour travailler **un analyseur lexical** qui lui **délivre** le **flot d'entrée** sous forme **d'unités lexicales** successives. Par défaut, **yacc** suppose que l'analyseur lexical disponible a été fabriqué par **lex**.

Pour cela, le programme produit par **yacc** comporte des appels de la fonction **yylex** (de **lex**) aux endroits où l'acquisition d'une **unité lexicale** est nécessaire.

Le schéma en dessous illustre les étapes de construction d'un analyseur syntaxique complet en utilisant **yacc et lex**.



Les commandes pour créer le fichier exécutable, **bas.exe** (l'analyseur syntaxique), sont :

- **yacc -d bas.y** # créer les fichiers **y.tab.h**, **y.tab.c**
- **lex bas.l** # créer le fichier **lex.yy.c**
- **gcc lex.yy.c y.tab.c -o bas.exe -lfl** # créer le fichier exécutable **bas.exe**

4.6.2 Structure d'un fichier source pour yacc :

Un fichier source pour **yacc** doit avoir un nom terminé par « **.y** ». Il se compose de trois sections, délimitées par le symbole « **%%** » :

```
%{
déclarations pour le compilateur C
%}
déclarations pour yacc
%%
 règles (productions + actions sémantiques)
%%
fonctions C supplémentaires
```

La partie « **déclarations pour le compilateur C** » est simplement recopiée dans le fichier à produire. La partie « **déclarations pour yacc** » sert pour spécifier les déclarations **des unités lexicales.**

Exemple:

```
%{
    char nom[256];
    int b = 50;
%}
%token nombre identificateur
```

La partie « **règles (productions + actions sémantiques)** », sert pour définir les productions de la grammaire du langage choisi.

Chaque production s'écrit sous la forme suivante :

```
non_terminal:
  corps_1      {action_sémantique_1}
  | …
  | corps_n   {action_sémantique_n}
  ;
```

Exemple:

```
terme :
 terme '*' facteur    { printf(" *"); }
| facteur
 ;
```

La partie «**fonctions C supplémentaires**» contient obligatoirement la fonction **main()** qui contient généralement la fonction **yyparse( )** qui est utilisée pour exécuter l'analyseur.

Exemple :

```
int main(void) {
 if (yyparse() == 0)
printf("Texte correct\n");
}
```

Remarque :

Dans un programme **yacc**, il faut écrire une fonction qui est appelée en cas d'erreur avec la spécification suivante :

void yyerror(char *message).

Exemple :

```
void yyerror(char *message) {
   printf(" <<< %s\n", message);
}
```

4.6.3 Exemple de Lex et Yacc :

Ce code combine **Lex** (analyseur lexical) et **Yacc** (analyseur syntaxique) pour analyser des **expressions mathématiques simples**. **Lex** identifie les **tokens** (nombres, identificateurs, opérateurs) à partir du texte d'entrée, tandis que **Yacc** structure ces tokens selon une **grammaire** pour valider les expressions.

```
%{
#include "syntaxe.tab.h"
extern char nom[];  /* chaîne de caractères partagée avec l'analyseur syntaxique */
%}
chiffre [0-9]
lettre [A-Za-z]
%%
[" "\t\n]                    { }
{chiffre}+                   { yylval = atoi(yytext); return nombre; }
{lettre}({lettre}|{chiffre})* { strcpy(nom, yytext); return identificateur; }
.                            { return yytext[0]; }
%%
int yywrap(void)
{ return 1; }
```

Fichier de spécification lex

```
%{
char nom[256];   /* chaîne de caractères partagée avec le fichier lex */
%}
%token nombre identificateur
%%
expression : expression '+' terme     { printf(" +"); }
| terme ;
terme : terme '*' facteur { printf(" *"); }
| facteur ;
facteur : nombre                      { printf(" %d", yylval); }
| identificateur                      { printf(" %s", nom); }
| '(' expression ')' ;
```

Fichier de spécification yacc

```
%%
void yyerror(char *s)
{
printf("<<< \n%s", s);
}
main()
{
if (yyparse() == 0)
printf(" Expression
    correcte\n");
}
```

4.6.4 Variables de YACC :

YACC utilise des variables pour désigner les non terminaux et les terminaux dans une grammaire.

- **$$** désigne l'attribut associé à la partie gauche d'une règle de production.
- **$i** désigne l'attribut associé au **ième** symbole d la partie droite d'une règle de production.
- La règle sémantique par défaut est {**$$=$1**}.
- La production vide est représentée par une alternative vide.

4.6.5 Les conflits dans YACC :

Yacc effectue une action par défaut en cas de conflit :

I) Conflits Décaler/Réduire: **Yacc** choisi un **décalage**.

II) Conflits de Réduire/Réduire: **Yacc** utilise la première règle de la liste. Il affiche également un message d'avertissement chaque fois qu'un conflit existe.

On peut spécifier dans un fichier **YACC** une grammaire **ambiguë**. Mais il faut spécifier les **associativités** et **priorités** des **opérateurs** dans la partie déclaration, pour lever l'ambiguïtés.

- Les associativités sont spécifiées par les mots clés **Left** (pour associativité gauche) ou **Right** (pour une associativité droite).
- Les **opérateurs** ainsi spécifiés auront des **priorités croissantes** selon **l'ordre** d'écriture.

Exemple :

- **%left '+','-'**
- **%left '*','/'**

Chapitre 5. Analyse syntaxique descendante

5.1 Introduction :

Dans **l'analyse syntaxique descendante**, l'arbre syntaxique est construit depuis la racine puis en descendant dans l'arbre.

Les méthodes descendantes sont **simples à implémenter**, mais la classe des grammaires non contextuelles, avec lesquelles on peut construire ces analyseurs, **n'est pas très grande**.

Nous allons étudier les deux versions de la méthode d'analyse prédictive **LL** :

- **L'analyse syntaxique LL(1).**
- **L'analyse par descente récursive.**

5.2 Analyse syntaxique LL(1) :

L'analyse LL est une **analyse syntaxique descendante** pour certaines grammaires non contextuelles, dites **grammaires LL**.

L'analyse LL analyse un mot d'entrée **de gauche à droite** (**Left to right** : c'est **le premier L de LL**) et en construit une **dérivation la plus à gauche** (**Leftmost derivation** : c'est **le deuxième L de LL**).

Une analyse LL est appelée analyse **analyse LL(1)** lorsqu'elle réalise une seule passe sur le mot d'entrée.

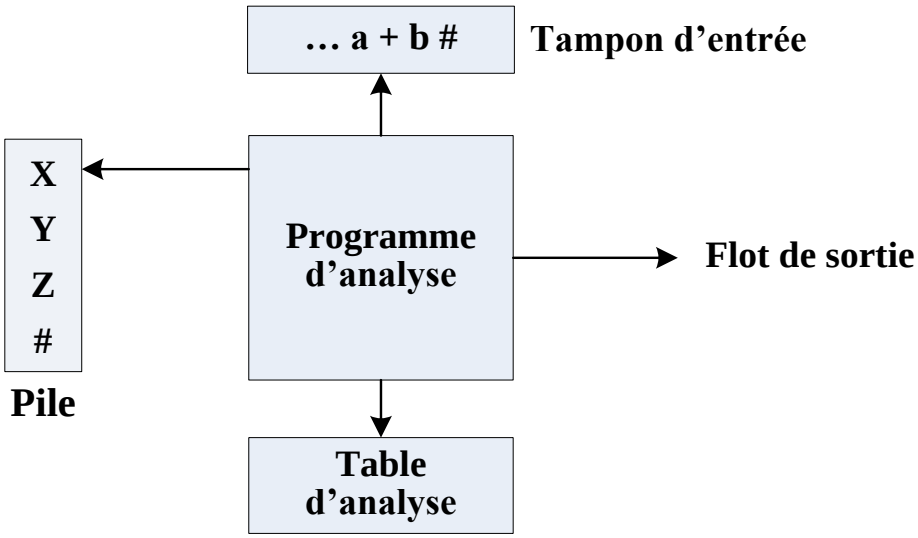
Une analyse LL est appelée **analyse LL(*k*)** lorsqu'elle réalise ***k*** passes sur le mot d'entrée.

Le nombre ***k*** est dans la majorité des cas égal à **1**, car pour **k>1** l'analyse devient **moins intéressante**. C'est pour cette raison que nous utiliserons uniquement l'analyse **LL(1)** dans la suite du cours.

5.2.1 Architecture d'un analyseur LL(1) :

L'analyseur LL(1) est composé de :

- Un **tampon d'entrée**, contenant la chaîne de caractères à analyser et doté de deux opérations : **lecture du caractère courant** et **passage au caractère suivant** ;
- Une **pile** sur laquelle poser les **terminaux** et **non terminaux** de la grammaire qui restent à analyser ;
- Une **table d'analyse** qui dit quelle règle utiliser (s'il y en a une) en fonction **des symboles au sommet de la pile** et du **caractère suivant**.



5.2.3 Construction de la table d'analyse LL(1) :

Soit ***G=<N,T,S,P>*** une grammaire non contextuelle (où ***N*** désigne l'ensemble des variables ou **symboles non terminaux**, ***T*** désigne l'ensemble des **symboles terminaux**, ***P*** l'ensemble des **règles**, ***S*** l'**axiome** de la grammaire).

Afin de construire **la table d'analyse**, on introduit les deux fonctions : ***Premier*** et ***Suivant*** (***First*** et ***Follow*** en anglas).

5.2.3.1 Fonctions : Premier et Suivant :

5.2.3.1.1 Fonction Premier :

Pour toute expression $\alpha \in (T \cup N)^+$, on définit ***Premier*** (α) comme étant l'ensemble des terminaux susceptibles de commencer un mot **dérivé** de **α.**

Plus formellement :

- $Premier(\alpha) = \{a \in T \mid \exists \beta \in (T \cup N)^*, \alpha \Rightarrow^* a\,\beta\}$.
- Si $\alpha \Rightarrow^* \varepsilon$ alors $\varepsilon \in Premier(\alpha)$.
- Si $\alpha = \varepsilon$ alors $Premier(\alpha) = \Phi$.

Calcul de l'ensemble Premier :

Algorithme Premier(X)

i) **SI** **X** est un terminal **ALORS** **Premier (X)= X ;**

ii) **SI** $X \rightarrow \varepsilon$ **ALORS** ajouter **ε** à **Premier (X) ;**

iii) **SI** $X \rightarrow Y_1Y_2\ldots Y_n$ **ALORS**

- **SI** $\varepsilon \notin$ **Premier (Y1)** **ALORS** on ajoute uniquement **Premier** (Y_1) à **Premier (X);**
- **SI** $\varepsilon \in$ **Premier** (Y_1)**,…, Premier**(Y_{i-1}) avec $2 \leq i \leq n$ **ALORS** ajouter **Premier**(Y_1)**,…, Premier**(Y_{i-1})**, Premier**(Y_i) **ε** exclu à **Premier (X)**

- **SI** $\varepsilon \in$ **Premier** (Y_1), **Premier** (Y_2),…, **Premier** (Y_n) **ALORS** ajouter **ε** à **Premier(X)**

Remarque :

Ces règles sont appliquées jusqu'à ce qu'aucun terminal ni ε ne puisse être ajouté aux ensembles Premier.

5.2.3.1.2 Fonction Suivant :

Pour toute expression $\alpha \in (T \cup N)^+$, on définit ***Suivant (α)*** comme étant l'ensemble des *terminaux* susceptibles de suivre un mot **dérivé** de α.

Plus formellement :

- $Suivant(\alpha) = \{a \in Premier(\gamma) \mid \exists \beta, \gamma \in (T \cup N)^*, S \Rightarrow^* \beta \alpha \gamma\}$.
- *Si* $\alpha = \varepsilon$ *alors* $Suivant(\alpha) = \Phi$.
- On ajoute aussi le symbole **'#' à tous les** $\alpha \in (T \cup N)^+ \mid \exists \beta \in (T \cup N)^*, S \Rightarrow^* \beta \alpha$, de façon à pouvoir indiquer la fin de la chaine d'entrée.

Calcul de l'ensemble SUIVANT

Algorithme SUIVANT(X)

i) Mettre **#** (qui est le marqueur de fin de l'entrée à analyser) dans **SUIVANT(S)** où **S** est **l'axiome** de la grammaire **;**

ii) S'il y a une production $A \rightarrow \alpha X \beta$ **ALORS** ajouter **PREMIER(β)** sauf **ε** à **SUIVANT(X) ;**

iii) S'il y a une production $A \rightarrow \alpha X$ ou une production $A \rightarrow \alpha X \beta$ avec $\varepsilon \in$ **PREMIER(β)** **ALORS** ajouter **SUIVANT(A) à SUIVANT(X);**

Remarque :

Ces règles sont appliquées jusqu'à ce qu'aucun terminal ni # ne puisse être ajouté aux ensembles **SUIVANT.**

Exemple 1:

Calculer les ensembles **Premier** et **Suivant** pour les non-terminaux de la grammaire dont les productions sont données ci-après : $S \rightarrow aSBA \mid \varepsilon$

$A \rightarrow aSb \mid b$

$B \rightarrow bB \mid \varepsilon$

	Premier	Suivant
S	a ε	# a b
A	a b	# a b
B	b ε	a b

Exemple2 :

Calculer les ensembles **Premier** et **Suivant** pour les non-terminaux de la grammaire dont les productions sont données ci-après **:** **S → ABSb | ε**

A → aBb | b

B → bB | cS | ε

	Premier	Suivant
S	a b ε	# a b
A	a b	b c a
B	b c ε	a b

5.2.3.2 Remplissage de la table d'analyse :

La **table d'analyse** est une **matrice à deux dimensions**, dont les **lignes** sont indicées par des ***Non-terminaux*** et les **colonnes** par **des *Terminaux***.

- **Donnée** : Une grammaire non-contextuelle **G**.
- **Résultat :** Une table d'analyse **M** pour **G**;

Les indices des lignes de **M** sont **les non-terminaux** de la grammaire ;

Les indices des colonnes sont **les terminaux** de la grammaire plus le caractère spécial # (# est le marqueur de fin du flot à analyser).

Algorithme de remplissage :

```
Pour toute règle de la forme X→α      Faire
        Pour tout a ∈ Premier(α)     Faire
                Ajouter X→α à la case d'indice M[X, a]
                Si ε ∈ Premier(α)    Alors
                        Pour tout b ∈ Suivant(X)      Faire
                                Ajouter X→α à la case d'indice M[X, b]
                        Fin pour
                Fin Si
        Fin pour
Fin pour
```

5.2.4 Grammaires LL(1)?

Une **grammaire** est LL(1) lorsque **sa table d'analyse ne comporte pas plusieurs règles de production** pour une **même case**. Dans ce cas la table d'analyse est dite **mono-définie.** Dans le cas contraire la table d'analyse est dite **multi-définie.**

Une grammaire **LL(1)** permet de faire une **analyse syntaxique descendante déterministe.** Au cours de toute dérivation, il existera au plus une possibilité pour remplacer un **non terminal** par une de ses **parties droites** selon le **caractère courant** de l'entrée à analyser.

Sans construire de la **table d'analyse**, on peut dire qu'une grammaire est **LL(1)** ou **non**, si elle a **les propriétés suivantes**:

S'il existe deux productions $A \rightarrow \alpha$ et $A \rightarrow \beta$ avec $(\alpha \neq \beta)$ dans une grammaire **LL(1)**, on a:

- Il **n'existe** pas de terminal ***a*** tel que **α** et **β** génèrent tous deux des chaînes qui commencent par ***a***;
- Au plus un de **α** et **β** peut générer la chaîne **ε**;
- Si **β ⇒* ε** alors **α ne peut pas générer** une chaîne dont le premier terminal est un élément de **Suivant(A)**.

Remarque :

Les trois conditions précédentes peuvent être résumées par la formule ci-après.

Une grammaire **G=<N,T,S,P>** est **LL(1)** si et seulement si pour toute paire de règles **A → α | β** on a :

$$\textbf{Premier}(\alpha.\textbf{SUIVANT}(A)) \cap \textbf{Premier}(\beta.\textbf{SUIVANT}(A)) = \varnothing$$

5.2.4 Algorithme d'analyse LL(1) :

```
Initialement la pile contient # et au dessus l'axiome de la grammaire;
répéter Soit X le symbole en sommet de pile et Soit a le symbole d'entrée
        courant
    Si X est un non terminal alors
        Si M[X, a]= X → Y1 ….Yn alors
            Dépiler X de la pile et Empiler Yn puis Yn-1
            puis…… puis Y1 dans la pile
        Sinon /*case vide dans la table*/
            Ecrire ('ERREUR')
        Finsi
    Sinon /*X est un terminal*/
(1) (2)
```

```
(1) (2)
        Si X = # alors
            Si a = # alors
                Ecrire ('ACCEPTER')
            Sinon
                Ecrire ('Erreur')
            Fin si
        Sinon /*X est un terminal ≠ #*/
            Si X = a alors
                Dépiler X de la pile et Avancer la lecture sur le symbole suivant.
            Sinon /*X est un terminal ≠ a*/
                Ecrire ('Erreur')
            Finsi
        Finsi
    Finsi
Fin Répéter
```

Exemple d'analyse LL(1) :

Considérer la grammaire **G = <{E,E',T,T',F},{+,*,(,),id},P,E>**:

E → TE'

E' → +TE' | ε

T → FT'

T' →* FT' | ε

F → (E) | i

Ensembles Premier et SUIVANT :

	Premier	SUIVANT
E	(i	#)
E'	+ ε	#)
T	(i	+ #)
T'	* ε	+ #)
F	(i	* + #)

Table d'analyse pour la grammaire précédente :

	+	*	(	)	I	#
E			**E → TE'**		**E → TE'**	
E'	**E' → +TE'**			**E' → ε**		**E' → ε**
T			**T → FT'**		**T → FT'**	
T'	**T' → ε**	**T' →* FT'**		**T' → ε**		**T' → ε**
F			**F → (E)**		**F → i**	

Selon la table d'analyse, qui est **mono-défini**, la grammaire précédente est LL(1).

Analyse de la chaine **i + i * i # :**

Contenu de la pile	Restant à analyser	Action
#E	i + i * i #	Dépiler(E) et Empiler(E'T)
#E'T	i + i * i #	Dépiler(T) et Empiler(T'F)
#E'T'F	i + i * i #	Dépiler(F) et Empiler(i)
#E'T'i	i + i * i #	Avancer
#E'T'	+ i * i #	Dépiler(T')
#E'	+ i * i #	Dépiler(E') et Empiler(E'T+)
#E'T+	+ i * i #	Avancer
#E'T	i* i #	Dépiler(T) et Empiler(T'F)
#E'T'F	i * i #	Dépiler(F) et Empiler(i)
#E'T'i	i * i #	Avancer
#E'T'	* i #	Dépiler(T') et Empiler(T'F*)
#E'T'F*	* i #	Avancer
#E'T'F	i #	Dépiler(F) et Empiler(i)
#E'T'i	i #	Avancer
#E'T'	#	Dépiler(T')
#E'	#	Dépiler(E')
#	#	« Chaine acceptée »

5.2.5 Transformation d'une grammaire :

- **Théorème** : Une grammaire **ambiguë**, ou **récursive à gauche** ou **non factorisée à gauche**, **n'est pas** une **grammaire LL(1)**.
- il **n'existe** malheureusement **pas** de méthode **pour rendre une grammaire LL(1)**:
  - **une grammaire ambiguë** est une grammaire **mal** conçue. Il faut tenter de refaire la conception de la grammaire (tenir compte de la priorité des opérateurs, associer le premier else au dernier if…).
  - **Eliminer la récursivité à gauche** si cela est nécessaire.
  - **Factoriser les règles de production à gauche** si cela est nécessaire.

5.2.5.1 Factorisation à gauche d'une grammaire :

Des productions non factorisées à gauche d'une grammaire du type :

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$ **où** $\alpha \neq \varepsilon$

Sont remplacées par les productions suivantes :

$A \rightarrow \alpha A' \mid \gamma$

$A' \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$

Exemple :

Soit la grammaire G1 ayant les règles de production suivantes :

S → aC | abD | aBc

Après factorisation à gauche, on obtient une grammaire G'1 équivalente à G1, dont les règles de production sont :

S → aS'

S' → C | bD | Bc

5.2.5.2 Elimination de la récursivité à gauche d'une grammaire :

Une grammaire est dite **récursive à gauche** si elle contient un non-terminal **A** tel que :

$$A \Rightarrow^{*} A\alpha$$

- Où α est une chaine quelconque.

5.2.5.2.1 Elimination de la récursivité à gauche immédiate :

Les règles sous la forme suivante :

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$ où β ne commence pas par **A**.

Sont remplacées par les productions :

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \varepsilon$

Exemple :

Soit la grammaire G1 ayant les règles de production suivantes :

E → E+T | T

T → T*F | F

F → (E) | id

Après l'élimination de la récursivité à gauche, on obtient une grammaire G'1 équivalente à G1, dont les règles de production sont :

E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | id

5.2.5.2.2 Elimination de la récursivité à gauche indirecte :

Grammaire sans cycle (dérivations $\mathbf{A} \Rightarrow^{+} \mathbf{A}$) et sans production vide ($A \rightarrow \varepsilon$)

Algorithme

Début

Ordonner les non-terminaux $\mathbf{A_1, A_2, \ldots, A_n}$;

Pour i:=1 **à** n **Faire**

Pour j:=1 **à** i-1 **Faire**

Remplacer chaque production de la forme $\mathbf{A_i \rightarrow A_j \gamma}$

Par les productions $\mathbf{A_i \rightarrow \delta_1\gamma | \delta_2\gamma | \ldots \delta_k\gamma}$

Où $\mathbf{A_j \rightarrow \delta_1 | \delta_2 | \ldots | \delta_k}$ sont les productions $\mathbf{A_j}$ courantes.

Fait ;

Eliminer les récursivités gauches immédiates des productions $\mathbf{A_i}$;

Fait ;

Fin

Exemple :

Eliminer la récursivité à gauche de la grammaire suivante :

S → Ab | b | Sb

A → a | AS | Sb

Ordre **S , A**

Etape 1: Elimination de la récursivité à gauche directe:

S → AbS' | bS'

S' → bS' | ε

Etape 2: Substitution

A → a | AS | AbS'b | bS'b

Elimination de la récursivité gauche directe :

A → aA' | bS'bA'

A' → SA' | bS'bA' | ε

5.3 Analyse syntaxique par descente récursive :

L'analyse syntaxique **par descente récursive** est une méthode d'analyse syntaxique **descendante** dans laquelle on exécute un ensemble **de procédures récursives** pour traiter la chaîne d'entrée.

Une **procédure** est associée à chaque **non-terminal** d'une grammaire.

Une analyse syntaxique **récursive LL(1)** se décompose en deux parties :

- **(i) construction de l'analyseur** : on écrit une procédure d'analyse pour chaque non-terminal
- **(ii) reconnaissance d'un mot** Une suite des appels de procédure **récursive** pour reconnaître un mot du langage.

5.3.1 Construction de l'analyseur :

Condition : la grammaire doit vérifier les conditions LL(1).

Les étapes de construction :

On ajoute la règle de production suivante : **Z → S#** où **S** est l'axiome de la grammaire et **#** le marqueur de fin de la chaine à analyser ;

A chaque non terminal de la grammaire correspond une procédure ;

5.3.2 Ecriture des procédures :

On utiliser les variables **tc** et **ts** pour désigner, respectivement, le **symbole courant** de la chaîne d'entrée à analyser et son **symbole suivant.**

Le traitement d'un membre droit d'une règle de production **A → α_i** se fera comme suit :

- **Chaque terminal** en partie droite est comparé au symbole courant de l'entrée :
  - **Si égalité** lire le symbole suivant
  - **Sinon** déclencher l'impression d'une erreur ;
- **Chaque non-terminal** en partie droite conduit à son appel.

Exemple :

Considérer la grammaire

G = <{S,A}, {a,b,c},S,P> suivante:

S → a A b | ε

A → c A | ab

On ajoute la règle suivante : **Z → S #**

Ensembles **PREMIER** et **SUIVANT** :

	PREMIER	SUIVANT
S	a ε	#
A	c a	b

La grammaire **G** précédente vérifie les conditions **LL(1)**

Ecriture des procédures :

```
Procédure Z( )
Début
   S( );
   Si tc = '#'
   Alors "Chaine syntaxiquement
   correcte"
   Sinon "Erreur"
   FinSi;
Fin.
```

```
Procédure S( )
Début
   Si tc = 'a'
   Alors
      tc = ts;
      A( );
      Si tc = 'b'
      Alors tc=ts
      Sinon "Erreur"
      FinSi;
   FinSi;
Fin.
```

```
Procédure A( )
Début
Si tc='c'
Alors
   tc = ts ;
   A( );
Sinon
```

5.3.3 Analyse :

L'analyse de la chaine : accabb# :

Contenu de la Pile	Restant à analyser	Action
Z	a c c a b b #	Appel S
ZS	a c c a b b #	Avancer ; Appel A
ZSA	c c a b b #	Avancer ; Appel A
ZSAA	c a b b #	Avancer ; Appel A
ZSAAA	a b b #	Avancer ; avancer ; retour A
ZSAA	b#	retour A
ZSA	b#	retour A
ZS	b#	avancer ; retour S
Z	#	"Chaine Correcte"

Analyse de la chaine : acaabb# :

Contenu de la Pile	Restant à analyser	Action
Z	a c a a b b #	Appel S
ZS	a c a a b b #	Avancer ; Appel A
ZSA	c a a b b #	Avancer ; Appel A
ZSAA	a a b b #	Avancer ; "Erreur"
ZSAA	a b b #	"Chaine incorrecte"

Chapitre 6. Analyse syntaxique ascendante

6.1 Introduction :

Les méthodes ascendantes cherchent à construire une **dérivation la plus à droite inverse** à partir de la chaîne de terminaux vers l'axiome de la grammaire.

De façon équivalente, **les méthodes ascendantes** cherchent à construire **un arbre syntaxique** à partir des **feuilles** en utilisant un **parcours inverse** en profondeur d'abord **de gauche à droite.**

6.2 Principe de l'analyse ascendante :

Le modèle par **décalage-réduction (shift-reduce)** est généralement utilisé dans l'analyse ascendante.

- **Décalage (shift)** : décaler le pointeur de lecture d'un symbole, sur la chaîne d'entrée.
- **Réduction (reduce)** : réduire une chaîne de symbole (terminaux et non terminaux) par un non terminal en utilisant une des règles de production de la grammaire. La chaîne se trouve à gauche du pointeur de lecture sur l'entrée et finissant sur ce pointeur.

Exemple :

Soit la grammaire **G** avec les règles de production suivantes :

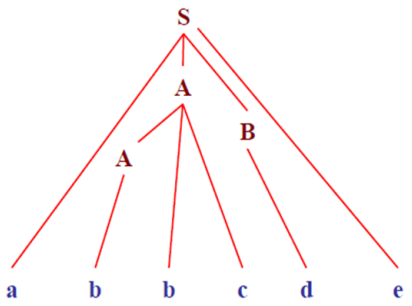
S → aABe

A → Abc| b

B → d

On veut analyser la chaîne **abbcde** de manière ascendante.

a b b c d e	décalage
a **b** b c d e	réduction par **A → b**
a **A** b c d e	décalage
a A **b** c d e	décalage
a A b **c** d e	réduction par **A → Abc**
a **A** d e	décalage
a A **d** e	réduction par **B → d**
a A **B** e	décalage
a A B **e**	réduction par **S → aABe**
S	"Chaîne accéptée"



Arbre syntaxique obtenu de manière ascendante

6.3 Analyse par la méthode LR :

L'analyse LR(K) est une technique efficace d'analyse syntaxique **ascendante** qui peut être utilisée pour analyser une **large classe de grammaire non contextuelle** :

- **Signification de LR(k) :**
  - **L** : Left to right parsing, parcours de l'entrée de la gauche vers la droite
  - **R** : Rightmost derivation in reverse, construire une dérivation droite inverse
  - **k** : k tokens to take an analysis decision. K indique le nombre de symboles de prévision utilisés pour prendre une décision d'analyse.

L'analyseur LR(K) est très intéressant pour les raisons suivantes :

- Les analyseurs LR peuvent être utilisé dans **de nombreux langages de programmation** et dans la **majorité des constructeurs d'analyseur syntaxiques**.
- La classe des grammaires qui peuvent être analysées par la méthode LR est un **sur-ensemble** strict de la classe des grammaires qui peuvent être analysées par les méthodes prédictives (grammaires LL).
- Un analyseur LR peut détecter **le plus tôt possible** une erreur de syntaxe.

L'inconvénient major de la méthode **LR** est la difficulté de produire à la main de tels analyseurs ; pour cela ils sont généralement construits par des générateurs d'analyse syntaxique comme Yacc par exemple qui utilise une des méthode LR qui est **LALR(1)**.

L'analyse LR se fait en construisant un automate AFD spécifiant les différents états d'avancement de l'analyse. Puis cet automate est transcrit en **table d'analyse.**

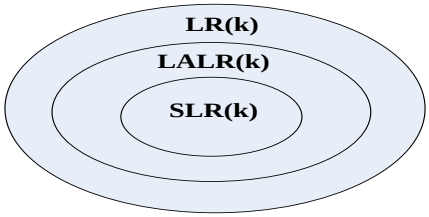
Les principales méthodes LR sont :

- **SLR(1) :** la plus simple à implémenter, mais la moins puissante des autres
- **LR(1)** : très puissante mais nécessite une mémoire importante
- **LALR(1)** : version affaiblie de LR(1) puissante et exécutable dans la majorité des grammaires des langage de programmation.

La classe des grammaires **LR(1)** est beaucoup plus grande que la classe des grammaires **SLR(1) et** plus grande que la classe des grammaires **LALR(1)**.

La classe des grammaires **LALR(1)** est plus grande que la classe des grammaires **SLR(1)**.

Un analyseur **LALR** utilise des tables qui ont strictement le même nombre d'états qu'un analyseur **SLR**.

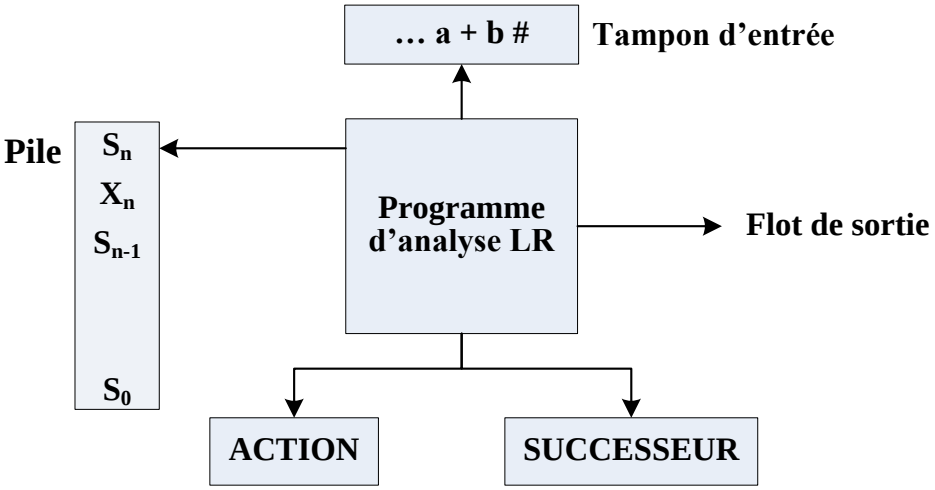


6.3.1 Programme d'analyse LR :

Le programme d'analyse LR est le **même** pour les différentes méthodes LR.

Il permet de faire une analyse syntaxique **déterministe** si la table d'analyse, associée à une grammaire, est **mono-définie**.

- Chaque méthode **LR** possède à **sa propre table d'analyse**, et chaque table d'analyse est organisée en deux parties : **ACTION** et **SUCCESSEUR**.
- **Le programme d'analyse LR** analyse **le flot d'entrée**, produit un **flot de sortie** en utilisant **la table d'analyse** et **une pile**.
- **L'architecture de l'analyseur LR** est décrit dans le schéma suivant :



Remarques :

La partie **ACTION** de la table d'analyse correspond à la sous table où les indices des colonnes sont **des terminaux**. La partie **SUCCESSEUR** (**fonction de transfert**) de la table d'analyse correspond à la sous table où les indices des colonnes sont **les non-terminaux**.

Les lignes de la table d'analyse correspondent aux **états** de l'automate AFD.

La pile est utilisée pour ranger des chaines de la forme **S0X1S1X2…XnSn** (sommet de pile à droite) où **Xi** est **un symbole de la grammaire** et **Si est un état**.

L'état en sommet de pile et **le symbole d'entrée courant** suffisent pour décider de l'action d'analyse à entreprendre.

6.3.2 Algorithme d'analyse LR :

- La pile contient au départ l'état initial de l'automate S_0 ;
- Soit S l'état en sommet de pile; et Soit a le symbole pointé par Ps;

```
Répéter indéfiniment
    Si ACTION[S,a]="DécalerT"  Alors
        empiler(a); empiler(T);      avancer Ps sur le prochain symbole;
    Sinon
        Si ACTION[S,a]="Réduire par A → α"  Alors
            dépiler 2*|α| symboles;
            Soit U le nouvel état en sommet de pile;
            empiler(A);          empiler(SUCCESSEUR[U,A]);
        Sinon
            Si ACTION[S,a]="Accepter"  Alors    retourner(Réussite());
            Sinon retourner(Echec());
            FinSi;
        FinSi;
    FinSi;
Fin;
```

Dans la suite, on va étudier chaque une des principales méthodes LR en détail :

Les trois principales méthodes LR sont :

- **SLR(1) :** la plus simple à implémenter, mais la moins puissante des autres
- **LR(1)** : très puissante mais nécessite une mémoire importante
- **LALR(1)** : version affaiblie de LR(1) puissante et exécutable dans la majorité des grammaires des langage de programmation

6.4 Analyse SLR(1) :

La construction d'une table d'analyse SLR (LR(0) ou SLR(1)) à partir d'une grammaire est la méthode la plus simple à implémenter (parmi les 3 méthodes **SLR (1), LALR (1) et LR(1)**) mais c'est la moins puissante du point de vue du nombre de grammaire pour lesquelles elle réussit.

La construction de la table d'analyse SLR(1) se fait comme suit:

- Construire l'automate d'états finis qui reconnait les préfixes viables de G (préfixes pouvant apparaître sur la pile);
- Transcrire cet automate en table ;
- Si la table d'analyse est mono-définie alors utiliser l'algorithme précédent pour effectuer l'analyse SLR(1).

La construction des tables d'analyse SLR, pour une grammaire donnée, est basée sur la **collection canonique d'items LR(0) (qui construisent les états de l'AFD)**.

Pour **construire** cette **collection**, on définit :

- **Une grammaire augmentée,**
- **Une fonction Fermeture,**
- **Une fonction Transition GOTO**

6.4.1 Construction de l'ensemble canonique d'items LR(0) :

Avant la construction de l'ensemble canonique d'items LR(0), il est important de comprendre ce que signifie le terme **item LR(0)**.

6.4.1.1 Item LR(0) :

Un item **LR(0)** d'une grammaire G est une production avec un point (.) repérant une position de sa partie droite .

Exemple :

La production **A → XYZ** fournit quatre items qui sont :

[A → .XYZ] , [A → X.YZ] , [A → XY.Z], [A → XYZ.]

La production A → ε fournit l'item :

[A → .]

Remarque

Un item indique la **quantité** de partie droite qui **a été reconnue à un instant donné** de l'analyse.

Par exemple, l'item **[A → X.YZ]** indique qu'on vient de reconnaitre une chaine dérivée de **X** et on attend de reconnaître une chaine dérivée de **YZ**.

6.4.1.2 Grammaire augmentée :

La grammaire initiale sera augmentée de la règle **S' → S** (où **S** est l'axiome de la grammaire) pour obtenir l'état initial de l'**AFD**

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

Soit **I** un ensemble d'items ;

- Placer chaque item de **I** dans **Fermeture(I)**;
- **Si [A→α.Bβ]** est dans **Fermeture(I)** et **B→γ** est une production : **Alors** ajouter l'item **[B→.γ]** à **Fermeture(I)** s'il n'y est pas ;
- Appliquer cette règle jusqu'à ce qu'aucun item ne puisse être ajouté à **Fermeture(I)**.

Exemple :

Considérer la grammaire G dont les productions sont données ci-après:

E → E + T | T

T → T * F | F

F → (E) | i

La grammaire est augmentée de la règle **E' → E** ;

Fermeture([E' → .E]) = {[E' → .E], [E→ .E+T], [E → .T], [T → .T*F], [T→ .F], [F → .(E)], [F → .i]}

6.4.1.4 Fonction de transition GoTo :

Soit **I** un ensemble d'items et **X** un symbole de la grammaire.

La fonction de transition GoTo (I, X) est définie comme suit :

GoTo (I, X) = fermeture [A→αX.β] tels que **[A →α.X β] ϵ I**.

Exemple :

En utilisant la grammaire de l'exemple précédent, Si **I= {[E'→E.], [E → E.+T]}**, alors la fonction **GoTo** est défini comme suit:

GoTo(I, +)= Fermeture ([E → E+.T]) = {[E → E+.T], [T →.T*F], [T →.F], [F →.(E)], [F →.i]}

6.4.1.5 Algorithme de construction de l'ensemble canonique d'items LR(0) :

Donnee : une grammaire **augmentée G0**

Résultat : une collection canonique d'ensembles d'items **LR(0)** (**C**) pour la grammaire **augmentée G0**

Méthode : on exécute la procédure suivante pour ajouter les transitions sur tous les symboles à partir de l'axiome :

```
Procédure Items(G0);
Début
    C = {Fermeture([S' → .S])};
    Répéter
        Pour chaque ensemble d'items I de C Faire
            Pour chaque symbole X de la grammaire tel que GOTO(I,X)
                                soit non vide et non encore dans C Faire
                ajouter GOTO(I,X) à C;
            Fin pour;
        Fin pour;
    Jusqu'à ce qu'aucun nouvel ensemble ne puisse être ajouté à C;
Fin.
```

Exemple :

Considérer la grammaire augmentée G0 pour la grammaire de l'exemple précédent :

E' → E

E → E + T | T

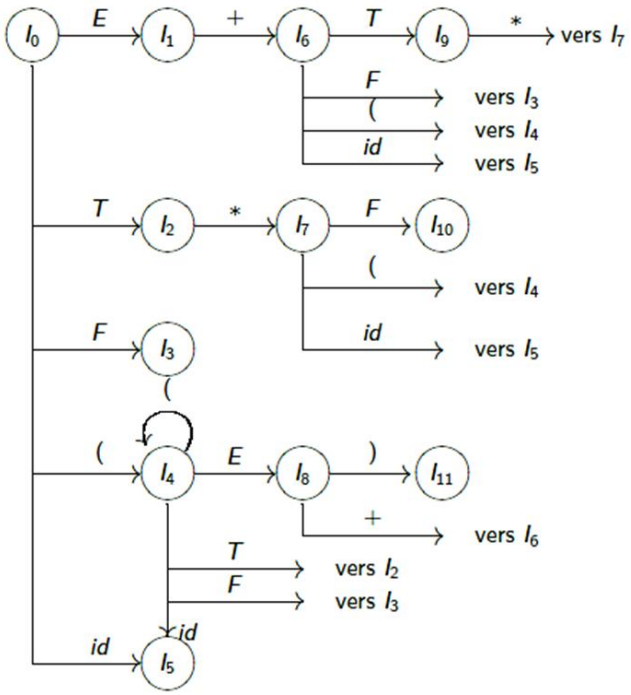
T → T * F | F

F → (E) | i

La collection canonique d'items LR(0) est la suivante:

- I_0 = {[E' → .E], [E → .E+T], [E → .T], [T→.T*F],[T → .F], [F → .(E)], [F → .i]}
- I_1 = GOTO(I_0, E) = {[E' → E.], [E → E.+T]}
- I_2 = GOTO(I_0, T) = {[E → T.], [T → T.*F]}
- I_3 = GOTO(I_0, F) = {[T → F.]}
- I_4 = GOTO(I_0, ( ) = {[F → (.E)], [E → .E+T], [E→ .T], [T → .T*F],[T → .F], [F→.(E)], [F→.i]}
- I_5 = GOTO(I_0, i) = {[F → i.]}
- I_6 = GOTO(I_1, +) = {[E → E+.T], [T→.T*F], [T→.F], [F → .(E)], [F→.i]}
- I_7 = GOTO(I_2, *) = {[T → T*.F], [F→.(E)], [F→ .i]}
- I_8 = GOTO(I_4, E) = {[F → (E.)], [E→E.+T]}
- GOTO(I_4, T) = I_2
- GOTO(I_4, F) = I_3
- GOTO(I_4, ( ) = I_4
- GOTO(I_4, i) = I_5
- I_9 = GOTO(I_6, T) = {[E → E+T.], [T→T.*F]}
- GOTO(I_6, F) = I_3
- GOTO(I_6, () = I_4
- GOTO(I_6, i) = I_5
- I_{10} = GOTO(I_7, F) = {[T → T*F.]}
- GOTO(I_7, () = I_4
- GOTO(I_7, i) = I_5
- I_{11} = GOTO(I_8,)) = {[F → (E).]}
- GOTO(I_8, +) = I_6
- GOTO(I_9, *) = I_7

Voici l'**AFD** construit pour la collection canonique d'items LR(0). Avec **I_0** est l'état initial.



6.4.2 Construction de la table d'analyse SLR (1) :

Algorithme :

Début

L'état **i** est construit à partir de **Ii**.

- La partie **ACTION** pour l'état **i** est déterminée comme suit:

Si [A→α.aβ] ϵ Ii et GoTo (Ii, a)= Ij (avec a ϵ T) **Alors**
Ajouter dj (décaler j) à ACTION [i, a]
Fin si
Si [A→α.] ϵ Ii (avec A≠S') **Alors**
pour tout a ϵ Suiv(A)
Ajouter r(k) à ACTION [i, a] (k est le numéro de la règle A→α)
Fin pour
Fin si
Si [S'→S.] ϵ Ii Alors maître « accepter » à ACTION [i, #]
Fin si

-La partie **SUCCESSEUR** pour l'état **i** est déterminée comme suit :

Si Goto (Ii, A)= Ij **Alors**
Ajouter j à SUCCESSEUR [i, A]
Fin si
- Toutes les entrées restantes dans la table sont mises à « ERREUR »
-**L'état initial** de l'analyseur est construit à partir de l'ensemble d'items contenant [S'→.S]

Fin ;

Exemple :

Considérer la grammaire augmentée G0 pour la grammaire de l'exemple précédent dont les productions numérotées sont données ci-après :

0) E' → E

1) E → E + T

2) E → T

3) T → T * F

4) T → F

5) F → (E)

6) F → i

	SUIVANT
E	#) +
T	#) + *
F	#) + *

La table d'analyse SLR(1) correspondant est :

	+	*	(	)	i	#	E	T	F
0			D 4		D 5		1	2	3
1	D 6					Accepter			
2	R (2)	D 7		R (2)		R (2)			
3	R (4)	R (4)		R (4)		R (4)			
4			D 4		D 5		8	2	3
5	R (6)	R (6)		R (6)		R (6)			
6			D 4		D 5			9	3
7			D 4		D 5				10
8	D 6			D 11					
9	R (1)	D 7		R (1)		R (1)			
10	R (3)	R (3)		R (3)		R (3)			
11	R (5)	R (5)		R (5)		R (5)			

La table est **mono-définie alors** la grammaire précédente est **SLR(1);**

6.4.3 Analyse SLR(1) :

Le tableau suivant montre comment analyser la chaine **"i+i*i#"** en utilisant la table d'analyse construit précédemment et l'algorithme d'analyse LR présenté dans la section **6.3.2.**

Pile	Restant à analyser	Action
0	i + i * i #	D 5
0 i 5	+ i * i #	R (6)
0 F 3	+ i * i #	R (4)
0 T 2	+ i * i #	R (2)
0 E 1	+ i * i #	D 6
0 E 1 + 6	i * i #	D 5
0 E 1 + 6 i 5	* i #	R (6)
0 E 1 + 6 F 3	* i #	R (4)
0 E 1 + 6 T 9	* i #	D 7
0 E 1 + 6 T 9 * 7	i #	D 5
0 E 1 + 6 T 9 * 7 i 5	#	R (6)
0 E 1 + 6 T 9 * 7 F 10	#	R (3)
0 E 1 + 6 T 9	#	R (1)
0 E 1	#	"Accepter"

6.4.4 Conflits dans l'analyse SLR(1) :

Deux types de **conflits existent** dans l'analyse **SLR(1):**

- **Conflit Décaler – Réduire :**

  Certains états peuvent contenir un **item** de la forme **[A →α.aβ]** et un item de la forme **[A →α.]**.

- **Conflit Réduire – Réduire :**

  Certains états peuvent contenir un item de la forme **[A →α.]** et un item de la forme **[B→β.]**. Avec **suivant (A) ∩ suivant (B) ≠Φ ;** Ils représentent deux situations de réduction par deux productions différentes.

Exemple :

- I_0 = {[E' → .E], [E → .E+T], [E → .T], [T→.i], [T → .(E)], [T → .ii[E]]}

- I_1 = GOTO(I_0, E) = {[E' → E.], [E → E.+T]}
- I_2 = GOTO(I_0, T) = {[E → T.]}
- I_3 = GOTO(I_0, i) = **{[T→i.],[T → i.i[E]]}**

  Ici il y a un **conflit Décaler - Réduire en items LR(0)**

- I_4 = GOTO(I_0, ( ) = {[F → (.E)], [E → .E+T], [E → .T], [T→.i], [T → .(E)], [T → .i[E]]}

6.5 Analyse LR(1) :

La méthode d'analyse LR(1) ou **LR canonique** est la technique la plus **générale** de construction de tables d'analyse LR pour une grammaire .

La classe des grammaires **LR(1)** est plus **grande** que celle des grammaires **SLR(1).**

La méthode LR(1) permet éviter un certain nombre d'actions invalides et de conflits en attachant plus d'informations à un état.

La construction des analyseurs LR (1), pour une grammaire donnée, est basée sur la **collection canonique d'items LR(1)**.

Pour **construire** cette **collection**, on définit:

- **une grammaire augmentée,**
- **une fonction Fermeture,**
- **une fonction Transition GOTO.**

6.5.1 Construction de l'ensemble canonique d'items LR(1) :

Avant la construction de l'ensemble canonique d'items LR(1), il est important de comprendre ce que signifie le terme **item LR(1)**.

6.5.1.1 Item LR(1) :

La forme générale d'un item **LR(1)** est **[A→α.β, a]** tel que : **A→αβ** est une règle de production de la grammaire et **a** est le **symbole de prévision** avec **a ϵ T ∪ {#}**.

Le symbole de prévision **a** n'a aucun effet dans un item de la forme **[A →α.β, a]** avec **β ≠ ε**,

Le symbole de prévision **a** implique « réduire par **A →α** uniquement lorsque le prochain symbole entré est **a** » dans un item de la forme **[A →α., a]**.

L'ensemble des symboles de prévision est un sous ensemble de **Suivant(A)**.

6.5.1.2 Grammaire augmentée :

La grammaire initiale sera augmentée de la règle **S' → S** (où **S** est l'axiome de la grammaire) pour obtenir l'état initial de l'**AFD**.

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

Soit **I** un ensemble d'items ;

- Placer chaque item de **I** dans **Fermeture(I)**;
- **Si [A→α.Bβ,a]** est dans **Fermeture(I)** et **B→γ** est une production: **Alors** ajouter l'item **[B→.γ,b]** avec **b ∈ PREMIER(βa)** à **Fermeture(I)** s'il n'y est pas ;
- Appliquer cette règle jusqu'à ce qu'aucun item ne puisse être ajouté à **Fermeture(I)**.

Exemple :

Considérer la grammaire G dont les productions sont données ci-après:

S → AA

A → aA

A → b

La grammaire est augmentée de la règle **S' → S** ;

Fermeture([S' → .S**,#]) = {[**S' → .S**,#], [**S→ .AA**,#], [**A → .aA**, a/b], [**A→ .b**, a/b]}**

6.5.1.4 Fonction de transition GoTo :

Soit **I** un ensemble d'items et **X** un symbole de la grammaire.

La fonction de transition GoTo (I, X) est définie comme suit : **GoTo (I, X) = fermeture [A→αX.β,a]** tels que **[A →α.X β,a] є I**.

Exemple :

En utilisant la grammaire de l'exemple précédent, Si **I=** {[S' → .S , #], [S → .AA , #], [A→.aA , a/b], [A→ .b , a/b]}, alors la fonction **GoTo(I,A)** est défini comme suit :

GoTo(I, A)= Fermeture ([S → A.A , #]) = {[S → A.A , #], [A → .aA , #], [A→ .b , #]}

6.5.1.5 Algorithme de construction de l'ensemble canonique d'items LR(1) :

- **Donnee** : une grammaire **augmentée G'**
- **Résultat** : une collection canonique d'ensembles d'items **LR(1) C** pour la grammaire **augmentée G'**
- Méthode : on exécute la procédure suivante pour ajouter les transitions sur tous les symboles à partir de l'axiome :

```
Procédure Items(G');
Début
    C = {Fermeture([S' → .S,#])};
    Répéter
        Pour chaque ensemble d'items I de C Faire
            Pour chaque symbole X de la grammaire tel que GOTO(I,X)
                                soit non vide et non encore dans C Faire
                ajouter GOTO(I,X) à C;
            Fin pour;
        Fin pour;
    Jusqu'à ce qu'aucun nouvel ensemble ne puisse être ajouté à C;
Fin.
```

Exemple :

Considérer la grammaire augmentée G' pour la grammaire de l'exemple précédent :

S' → S

S → AA

A → aA

A → b

La collection canonique d'items LR(1) est la suivante :

- I_0 = {[S' → .S , #], [S → .AA , #], [A→.aA , a/b], [A→ .b , a/b]}
- I_1 = GOTO(I_0, S) = {[S' → S. , #]}
- I_2 = GOTO(I_0, A) = {[S → A.A , #], [A → .aA , #], [A→ .b , #]}
- I_3 = GOTO(I_0, a) = {[A → a.A , a/b], [A → .aA , a/b], [A → .b , a/b]}
- I_4 = GOTO(I_0, b) = {[A → b. , a/b]}
- I_5 = GOTO(I_2, A) = {[S → AA. , #]}
- I_6 = GOTO(I_2, a) = {[A → a.A , #], [A → .aA , #], [A→ .b , #]}
- I_7 = GOTO(I_2, b) = {[A → b. , #]}
- I_8 = GOTO(I_3, A) = {[A → aA. , a/b]}
- GOTO(I_3, a) = I_3
- GOTO(I_3, b) = I_4
- I_9 = GOTO(I_6, A) = {[A → aA. , #]}
- GOTO(I_6, a) = I_6
- GOTO(I_6, b) = I_7

6.5.2 Construction de la table d'analyse LR (1) :

Algorithme :

Début

L'état **i** est construit à partir de **Ii**.

- La partie **ACTION** pour l'état **i** est déterminée comme suit:

Si [A→α.aβ,b] ϵ Ii et GoTo (Ii, a)= Ij (avec a,b ϵT) **Alors**
  Ajouter dj (décaler j) à ACTION [i, a]
Fin si
Si [A→α.,a] ϵ Ii (avec A≠S') **Alors**
  Ajouter r(k) à ACTION [i, a] (k est le numéro de la règle A→α)
Fin si
Si [S'→S.,#] ϵ Ii Alors maître « accepter » à ACTION [i, #]
Fin si

-La partie **SUCCESSEUR** pour l'état **i** est déterminée comme suit :

Si Goto (Ii, A)= Ij **Alors**
  Ajouter j à SUCCESSEUR [i, A]
Fin si
- Toutes les entrées restantes dans la table sont mises à « ERREUR »
-**L'état initial** de l'analyseur est construit à partir de l'ensemble d'items contenant [S'→.S,#]

Fin ;

Exemple :

Considérer la grammaire augmentée G' pour la grammaire de l'exemple précédent dont les productions numérotées sont données ci-après :

0) S' → S

1) S → AA

2) A → aA

3) A → b

La table d'analyse LR(1) correspondant est :

	a	b	#	S	A
0	D 3	D 4		1	2
1			Accepter		
2	D 6	D 7			5
3	D 3	D 4			8
4	R (3)	R (3)			
5			R (1)		
6	D 6	D 7			9
7			R(3)		
8	R (2)	R (2)			
9			R(2)		

La table est mono-définie donc la grammaire précédente est LR(1).

6.5.3 Analyse LR(1) :

Les tableaux suivants montrent comment analyser les chaines **"aaab#"** et **"aabb#"** en utilisant la table d'analyse construit précédemment et l'algorithme d'analyse LR présenté dans la section **6.3.2.**

- Analyse de la chaine **« aaab#» :**

Pile	Restant à analyser	Action
0	aaab#	D 3
0 a 3	aab#	D 3
0a3a3	ab#	D 3
0a3a3a3	b#	D 4
0a3a3a3b4	#	"Erreur"

- Analyse de la chaine **« aabb# »**

Pile	Restant à analyser	Action
0	aabb#	D 3
0 a 3	abb#	D 3
0a3a3	bb#	D 4
0a3a3b4	b#	R(3)
0a3a3A8	b#	R(2)
0a3A8	b#	R(2)
0A2	b#	D7
0A2b7	#	R(3)
0A2A5	#	R(1)
0S1	#	"chaîne acceptée"

6.5.4 Conflits dans l'analyse LR(1) :

Deux types de **conflits existent** dans l'analyse **LR(1):**

- **Conflit Décaler – Réduire :** Certains états peuvent contenir un **item** de la forme **[A → β.aγ,b]** et un item de la forme **[A →α.,a]**.
- **Conflit Réduire – Réduire :** Certains états peuvent contenir un item de la forme **[A →α.,a]** et un item de la forme **[B→β.,a]**. Ils représentent deux situations de réduction par le caractère **a** commun aux suivants de **A** et de **B**.

6.6 Analyse LALR(1) :

La méthode d'analyse LALR(1) (Look Ahead LR) est un **très bon compromis** entre les analyses **SLR(1)** et **LR(1)**.

Un analyseur **LALR(1)** utilise des tables qui ont **strictement le même nombre d'états** qu'un analyseur **SLR(1)**, mais la classe des grammaire **LALR(1)** est **plus grande** que la classe des grammaires **SLR(1).**

La méthode d'analyse **LALR(1)** est largement utilisée dans la pratique, par exemple le générateur de compilateur **YACC** utilise la méthode **LALR(1)** pour la partie syntaxique du compilateur.

La construction de la table d'analyse **LALR(1)** est **basé sur** la collection d'ensembles d'items **LR(1)** en regroupant les items **LR(1)** ayant le même **cœur** . Sachant que : **item LR(1)=[cœur, symbole de prévision].**

La table d'analyse LALR(1) est construite à partir de la collection des ensembles d'items **fusionnés**.

Remarque:

Il existe une autre méthode **plus efficace** pour construire des tables **LALR (1)**, basée sur les items **LR (0)** de la méthode **SLR (1)**. **Mais** dans **ce cours** nous utiliserons la méthode qui est basée sur les items **LR (1)** car elle est **plus simple**.

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

Début

I. Construire la collection d'ensembles d'items LR(1) **C =** $\{I_0, I_1, \ldots, I_n\}$.

II. 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.

III. Soit **C' =** $\{J_0, J_1, \ldots, J_m\}$ la nouvelle collection d'items;
L'opération **GOTO** est obtenue de la façon suivante :
Si $J = I_0 \cup I_1 \cup \ldots \cup I_s$ **Alors**
$K = \cup$ GOTO(I_i, X) , i=1…s;
GOTO(J,X) = K;
FinSi;

Exemple :

Considérer la grammaire augmentée G' pour la grammaire de l'exemple précédent :

S' → S

S → AA

A → aA

A → b

La collection canonique d'items LR(1) est la suivante:

- I_0 = {[S' → .S , #], [S → .AA , #], [A→.aA , a/b], [A→ .b , a/b]}
- I_1 = GOTO(I_0, S) = {[S' → S. , #]}
- I_2 = GOTO(I_0, A) = {[S → A.A , #], [A → .aA , #], [A→ .b , #]}
- I_3 = GOTO(I_0, a) = {[A → a.A , a/b], [A → .aA , a/b], [A → .b , a/b]}
- I_4 = GOTO(I_0, b) = {[A → b. , a/b]}
- I_5 = GOTO(I_2, A) = {[S → AA. , #]}
- I_6 = GOTO(I_2, a) = {[A → a.A , #], [A → .aA , #], [A→ .b , #]}
- I_7 = GOTO(I_2, b) = {[A → b. , #]}
- I_8 = GOTO(I_3, A) = {[A → aA. , a/b]}
- GOTO(I_3, a) = I_3
- GOTO(I_3, b) = I_4
- I_9 = GOTO(I_6, A) = {[A → aA. , #]}
- GOTO(I_6, a) = I_6
- GOTO(I_6, b) = I_7

Regroupement les items LR(1) qui ont le même cœur:

- Les états I_3 et I_6 ayant le même cœur:

  I_3 = {[A → a.A , **a/b**], [A → .aA , **a/b**], [A → .b , **a/b**]}

  I_6 = {[A → a.A , #], [A → .aA , #], [A→ .b , #]}

  seront regroupés en: I_{36} = {[A → a.A , **a/b/#**], [A → .aA , **a/b/#**], [A → .b , **a/b/#**]}

- Les états I_4 et I_7 ayant le même cœur:

  I_4 = {[A → b. , **a/b**]}

  I_7 = {[A → b. , **#**]}

  seront regroupés en: I_{47} = {[A → b. , **a/b/#**]}

- Les états I_8 et I_9 ayant le même cœur:

  I_8 = {[A → aA. , **a/b**]}

  I_9 = {[A → aA. , **#**]}

  seront regroupés en: I_{89} = {[A → aA. , **a/b/#**]}

La nouvelle collection canonique d'items LR(1) C' est:

- I_0 = {[S' → .S , #], [S → .AA , #], [A→.aA , a/b], [A→ .b , a/b]}
- I_1 = GOTO(I_0, S) = {[S' → S. , #]}
- I_2 = GOTO(I_0, A) = {[S → A.A , #], [A → .aA , #], [A→ .b , #]}
- I_{36} = GOTO(I_0, a) = {[A → a.A , a/b/#], [A → .aA , a/b/#], [A → .b , a/b/#]}

- I_{47} = GOTO(I_0, b) = {[A → b. , a/b/#]}
- I_5 = GOTO(I_2, A) = {[S → AA. , #]}
- GOTO(I_2, a) = I_{36}
- GOTO(I_2, b) = I_{47}
- I_{89} = GOTO(I_{36}, A) = {[A → aA. , a/b/#]}
- GOTO(I_{36}, a) = I_{36}
- GOTO(I_{36}, b) = I_{47}

Table d'analyse LALR(1) :

	a	b	#	S	A
0	D 36	D 47		1	2
1			Accepter		
2	D 36	D 47			5
36	D 36	D 47			89
47	R(3)	R(3)	R(3)		
5			R(1)		
89	R(2)	R(2)	R(2)		

La table est mono-définie donc la grammaire précédente est LALR(1).

6.6.2 Analyse LALR(1) :

Les tableaux suivants montrent comment analyser les chaines **"aaab#"** et **"aabb#"** en utilisant la table d'analyse construit précédemment et l'algorithme d'analyse LR présenté dans la section **6.3.2.**

0) S' → S

1) S → AA

2) A → aA

3) A → b

- Analyse de la chaine **« aaab#» :**

Pile	Restant à analyser	Action
0	aaab#	D 36
0a36	aab#	D36
0a36a36	ab#	D36
0a36a36a36	b#	D47
0a36a36a36b47	#	R(3)
0a36a36a36A89	#	R(2)
0a36a36A89	#	R(2)
0a36A89	#	R(2)
0A2	#	"Erreur"

- Analyse de la chaine **« aabb#»** :

Pile	Restant à analyser	Action
0	aabb#	D 36
0a36	abb#	D36
0a36a36	bb#	D47
0a36a36b47	b#	R(3)
0a36a36A89	b#	R(2)
0a36A89	b#	R(2)
0A2	b#	D47
0A2b47	#	R(3)
0A2A5	#	R(1)
0S1	#	"chaîne acceptée"

6.6.3 Conflits dans l'analyse LALR(1) :

Pendant la construction des tables **LALR(1)** à partir des tables **LR(1)** il y a un risque de conflit :

- **Conflit Décaler – Réduire :** La fusion de deux états ne peut entraîner de nouveaux conflits **Décaler - Réduire absents** des états **LR(1)** car les états fusionnés ont le même cœur.
- **Conflit Réduire – Réduire :** Le seul cas de conflit qui peut se produire durant la fusion des états **LR(1)**. C'est pour cette raison qu'il existe des grammaires **LR(1)** qui **ne sont pas LALR(1).**

Chapitre 7. Traduction dirigée par la syntaxe

7.1 Introduction :

Le compilateur possède trois étapes d'analyse : l'analyse lexicale, syntaxique et **sémantique**. L'étape d'analyse **sémantique** permet d'ajouter des informations à **l'arbre syntaxique** en construisant **l'arbre syntaxique décoré**. Pour cela, deux mécanismes sont mis en place :

- Le premier est utilisé pour les données et il s'appel : **définition dirigée par la syntaxe.**
- Le deuxième mécanisme est utilisé pour les calculs, et il s'appel : **traduction dirigée par la syntaxe**.

7.2 Définition dirigée par la syntaxe :

Une **Définition dirigée par la syntaxe (SDD)** est une formalisation qui associe des informations sémantiques à une grammaire pour décrire des propriétés ou des actions relatives à un langage de programmation. Elle combine une **grammaire hors contexte** avec des **attributs** et des **règles sémantiques** pour représenter à la fois la structure syntaxique et les significations associées.

7.2.1 Grammaire attribuée :

Les grammaires attribuées sont **des grammaires non contextuelles** qui ont été déjà utilisées dans **l'analyse syntaxique en ajoutant** quelques informations supplémentaires aux symboles (terminaux et non terminaux) appelés **attributs**.

Chaque **symbole** de la grammaire **peut** avoir des **attributs**.

Si le symbole **X** n'apparaît qu'une seule fois dans une règle de production, alors **X.a** désigne l'attribut **a** du symbole **X**.

Si le symbole **X** apparaît plusieurs fois dans une règle de production, alors on note **X** le symbole en partie gauche, et on note $\mathbf{X_1, X_2, \ldots X_n}$, les occurrences de **X** en partie droite.

Les valeurs des attributs de chaque règle de production sont calculées en utilisant un ensemble d'actions.

Exemple :

Soit la grammaire usuelle des expressions arithmétiques :

L → E

E → E+T | T

T → T∗F | F

F → (E) | id

Voici la grammaire attribuée qui permet de calculer la valeur d'une expression arithmétique:

E.val=E1.val+T.val

E.val=T.val

T.val=T1.val*F.val

T.val=F.val

F.val=E.val

F.val=id.vallex

Chaque **non-terminal** de la grammaire a un attribut appelé **val**.

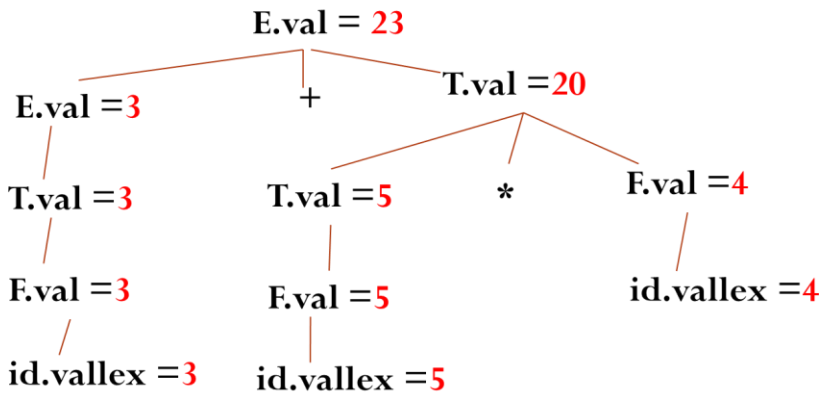
Le **terminal** id a un attribut appelé **vallex**, qui a une valeur entière rendu par l'analyseur lexical.

7.2.2 Arbre syntaxique décoré :

Un **arbre syntaxique décoré** est une version enrichie d'un arbre syntaxique abstrait. Il représente non seulement la structure syntaxique du code source, mais aussi des informations supplémentaires liées à l'analyse sémantique, telles que les types des variables, les valeurs calculées, les symboles ou autres métadonnées nécessaires pour le traitement ultérieur.

Exemple :

La figure suivante montre l'arbre syntaxique décoré pour la chaîne d'entrée : **3+5*4**.



7.2.3 Types d'attributs :

Il existe deux types d'attributs :

- **Attribut synthétisé :**
  - Un attribut synthétisé est associé au symbole en partie gauche d'une règle de production et sa valeur est calculée en se basant sur les valeurs des attributs des symboles de la partie droite.
  - Dans l'arbre syntaxique décoré, l'attribut synthétisé est associé à un nœud et sa valeur est calculée en se basant sur les valeurs des attributs de ses fils.
- **Attribut hérité :**

- Un attribut hérité est associé à un symbole de la partie droite d'une règle de production et sa valeur est calculé en se basant sur la valeur de l'attribut du non terminal de la partie gauche et des valeurs des attributs des autres symboles de la partie droite.
- Dans l'arbre syntaxique décoré, la valeur d'un attribut hérité est calculée en utilisant des valeurs des attributs du nœud père et des nœuds frères.

Exemple :

Attribut synthétisé :

L'attribut **val** dans l'exemple précédent est un attribut synthétisé.

Attribut hérité :

Soit la grammaire suivante :

E → T E'

E' → + T E' | T E' | #

T → F T'

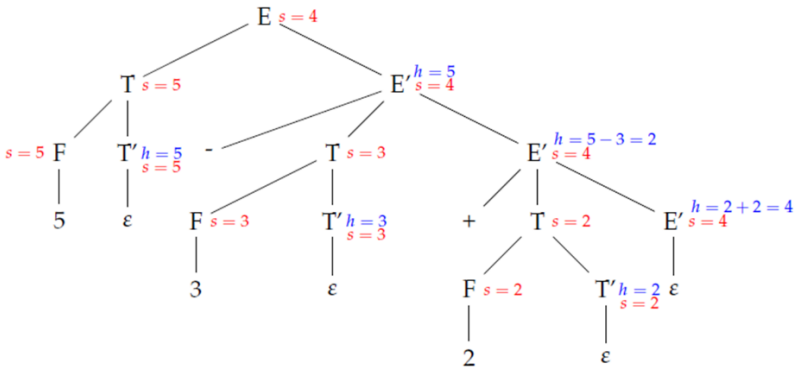
T' → * F T' |/ F T' | #

F → (E) | n

Voici la grammaire attribuée en utilisant un attribut synthétisé **s** et un attribut hérité **h.**

règle			action sémantique	
E	→	T E′	$E'.h = T.s$	$E.s = E'.s$
E′	→	+ T E′$_1$	$E'_1.h = E'.h + T.s$	$E'.s = E'_1.s$
E′	→	- T E′$_1$	$E'_1.h = E'.h - T.s$	$E'.s = E'_1.s$
E′	→	ε		$E'.s = E'.h$
T	→	F T′	$T'.h = F.s$	$T.s = T'.s$
T′	→	* F T′$_1$	$T'_1.h = T'.h \times F.s$	$T'.s = T'_1.s$
T′	→	/ F T′$_1$	$T'_1.h = T'.h \div F.s$	$T'.s = T'_1.s$
T′	→	ε		$T'.s = T'.h$
F	→	(E)		$F.s = E.s$
F	→	n		$F.s = n$

La figure suivante montre l'arbre syntaxique décoré pour la chaîne d'entrée : **3-3+2**.



Remarques :

On peut écrire une définition dirigée par la syntaxe d'une grammaire non contextuelle avec des attributs synthétisés seulement, mais on risque de perdre le sens des règles et des attributs dans certaines grammaire.

Les attributs hérités sont utilisés dans les cas suivantes :

- Déclaration précoce des variables ;
- Identification de type des variables ;
- Désignation des positions droites ou gauches par rapport à un opérateur d'affectation ;
- …..

7.2.4 Graphe de dépendances :

Le graphe de dépendances sert à identifier l'ordre d'évaluation des valeurs d'attributs **hérités** et/ou **synthétisés** dans un arbre syntaxique.

L'arbre syntaxique décoré est utilisé pour calculer les valeurs des attributs, et le graphe de dépendances est utilisé pour déterminer comment calculer ces valeurs.

Pour les attributs **synthétisés**, les attributs sont évalués par ordre ascendant (de fils en père).

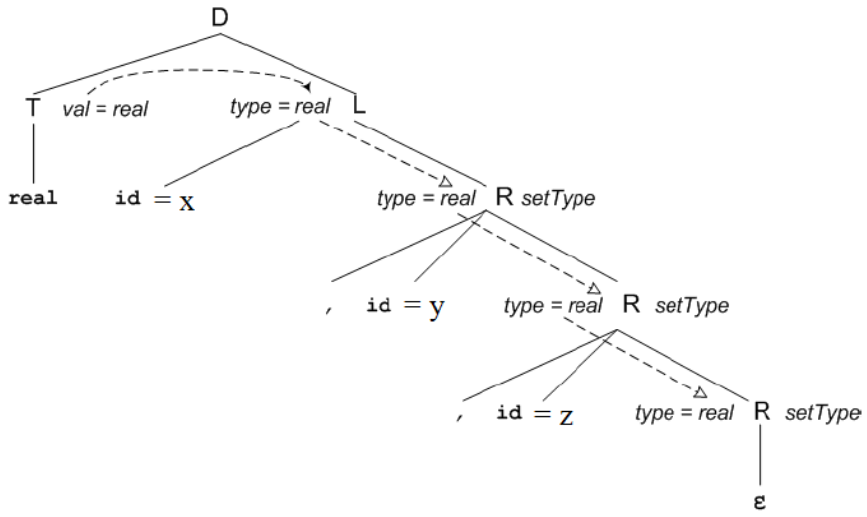
Pour les attributs **hérités**, les attributs sont évalués par ordre descendant (de père en fils) ou au diagonal (de frère en frère).

Exemple :

Voici la grammaire des déclarations en langage C et sa définition dirigée par la syntaxe.

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

Voici le graphe de dépendances pour la phrase : **real x, y, z :**



7.2.5 Définition S-attribuée et Définition L-attribuée :

7.2.5.1 Définition S-attribuée :

Une définition dirigée par la syntaxe est dite **S-attribuée** si elle n'ayant que **des attributs synthétisés**.

7.2.5.2 Définition L-attribuée :

Une définition dirigée par la syntaxe est dite **L-attribuée** si tout attribut **hérité** d'un symbole $\mathbf{X_j}$ de la partie droite d'une production de la forme $\mathbf{A \rightarrow X_1 X_2 \ldots\ldots X_n}$ ne dépend que :

- Des attributs hérités du symbole **A** en partie gauche.
- Des attributs des symboles $\mathbf{X_1 X_2 \ldots\ldots X_{j-1}}$ en partie droite.

7.3 Traduction dirigée par la syntaxe :

La **Traduction dirigée par la syntaxe (Syntax-Directed Translation : SDT)** est une technique basée sur les définitions dirigée par la syntaxe (**SDD**), mais orientée vers la **génération de sorties** (comme du code intermédiaire, des instructions machine ou d'autres représentations). Elle enrichit les productions syntaxiques avec des **actions sémantiques** (du code exécuté lors de l'analyse syntaxique).

7.3.1 Langages intermédiaires :

Il existe plusieurs langages intermédiaires :

- **Notation post-fixée**
- **Triplets**
- **Quadruplets**
- **…etc.**

7.3.1.1 Notation post-fixée :

La notation postfixée d'une expression est définie comme suit :

- Si une expression **E** est une constante ou une variable, sa notation postfixée est simplement **E**.
- Si **E** est une expression de la forme **E1 op E2** où **op** est un opérateur binaire, alors la notation postfixée de **E** est donnée par : **E1′ E2′ op** où **E1′** et **E2′** représentent les notations postfixées respectives de **E1** et **E2**.
- La notation postfixée de **(E)** est la même que la notation postfixée de **E**.

Exemple :

La notation postfixée pour l'expression : **b*(a−c)*a** est : **b a c − ∗ a ***

7.3.1.2 Triplet :

Un triplet est une structure composée de trois champs **:(op, Arg1, Arg2)**, où **op** est l'opération effectuée entre Arg1 et Arg2. Le résultat de cette opération est identifié par le numéro du triplet dans le code intermédiaire généré.

Exemple :

Représentation sous forme de triplets de l'expression : **a= b * (- c) + b * (- c)**

(0) (- , c ,)

(1) (* , b , (0))

(2) (- , c ,)

(3) (* , b , (2))

(4) (+ , (1) , (3))

(5) (= , a , (4))

7.3.1.3 Quadruplets:

Les quadruplets, largement utilisés dans la compilation des langages de programmation de haut niveau, représentent un format de code intermédiaire à trois adresses. Ils ont la structure suivante : **(op, arg1, arg2, cible)**.

Pour les instructions à opérateur unaire, la forme devient : **(op, arg1 , , cible)**

Les instructions de branchement sont représentées ainsi : **(code-br , , , label)**

Exemple 1 :

Représentation sous forme de quadruplets de l'expression **a = b * (-c) + b * (-c) :**

(0) (- , c , , T1)

(1) (* , b , T1 , T2)

(2) (- , c , , T3)

(3) (* , b , T3 , T4)

(4) (+ , T2 , T4 , T5)

(5) (= , T5 , , a)

Exemple 2:

Représentation sous forme de quadruplets du code suivant :

```
 int b[10];
for (int i = 0; i < 10; i++) {
    b[i] = i * i;
        }
```

Est:

```
t1 := 0            /* Initialisation de i */
t2 := b            /* Initialisation d'un pointeur sur b[i] */
L1:  if t1 >= 10
goto L2            /* Saut conditionnel */
t3 := t1 * t1      /* Calcul du carré de i */
*t2 := t3          /* Déréférencement, affectation dans b[i] */
t1 := t1 + 1       /* Incrémentation de i */
t2 := t2 + 4       /* Passage à la case b[i] suivante; (supposition pour une architecture 32 bits) */
goto L1            /* Répétition de la boucle */
L2:
```

Ou bien :

(:=, 0, , t1)

(:=, b, , t2)

L1:

(code-br, t1, 10, L2)

(*, t1, t1, t3)

(store, t3, , t2)

(+, t1, 1, t1)

(+, t2, 4, t2)

(code-br, , , L1)

L2:

7.3.2 Schéma de traduction :

Un schéma de traduction (**en anglais, translation scheme**) est un outil utilisé dans les compilateurs et les interprètes pour décrire comment traduire ou transformer un programme source en une représentation cible ou intermédiaire. Il combine une grammaire (définissant la structure syntaxique) avec des actions sémantiques (instructions ou code à exécuter lors de l'analyse syntaxique).

Dans un **schéma de traduction dirigé par la syntaxe (STDS)**, si une production a la forme suivante : **S → αX {action} Yβ**

L'**action** est exécutée après que le **sous-arbre syntaxique** dérivé de **X** a été complètement parcouru (en suivant un **parcours en profondeur**) et avant que le sous-arbre dérivé de **Y** soit traité.

7.3.3 Conception d'un schéma de traduction :

Lors de la conception d'un **schéma de traduction**, certaines restrictions doivent être respectées. Ces restrictions garantissent que la valeur d'un attribut est disponible au moment où une action s'y réfère.

Le cas le plus simple se présente lorsque seules des attributs synthétisés sont utilisés. Dans ce contexte :

Le schéma de traduction peut être construit en associant une action d'affectation à chaque règle sémantique.

Cette action est généralement placée à la fin de la partie droite de la production pour garantir que les attributs nécessaires ont été calculés.

Exemple :

T→ T1 * F { T.val := T1.val x F.val}

Lorsqu'on utilise à la fois des **attributs hérités** et des **attributs synthétisés**, il est nécessaire de respecter des règles précises pour garantir que les valeurs des attributs soient disponibles au bon moment. Voici les principales conditions :

Un **attribut hérité** d'un symbole situé dans la partie droite d'une production doit être calculé par une action placée **avant** ce symbole.

Une action ne doit pas référencer un **attribut synthétisé** d'un symbole qui apparaît **à droite** de l'action dans la production.

Un **attribut synthétisé** du non-terminal de gauche ne peut être calculé qu'après que tous les attributs dont il dépend ont été définis. En général, l'action qui calcule cet attribut est placée **à la fin** de la production.

Exemple : voici l'exemple suivant : **S → A1 A2 { A1.h := 1; A2.h := 2 }**

A → a { imprimer(A.h) }

Problème :

Dans cet exemple, lors du parcours en profondeur de l'arbre syntaxique pour la chaîne **aa**, l'attribut **hérité A.h** de la production **A → a** n'est pas défini au moment où la commande imprimer(A.h) est exécutée.

Pourquoi ?

Les **actions { A1.h := 1; A2.h := 2 }** qui définissent les attributs hérités des deux symboles **A1** et **A2** sont placées **après** les symboles dans la partie droite de la production **S → A1 A2**. Par conséquent, au moment où le nœud correspondant à **A1** ou **A2** est parcouru, leur attribut hérité **A.h** n'a pas encore été initialisé.

Solution : **S → { A1.h := 1 } A1 { A2.h := 2 } A2**

Ainsi, lorsque la commande imprimer(**A.h**) est exécutée dans **A→a**, l'attribut hérité **A.h** aura déjà une valeur définie.

Exemple :

Dans ce qui suit, on introduit le schéma de traduction pour les opérations arithmétiques en utilisant les quadruplets : **S → E #**

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

Des **quadruplets** sont générés pour chaque opération arithmétique, et des variables temporaires **(ti)** sont utilisées pour stocker les résultats intermédiaires :

Production	Règle sémantique	Traduction
S → E #	S.val := E.val	
E → E + T	E.val := E1.val + T.val	(+, E1.val,T.val, t1)
E → E - T	E.val := E1.val - T.val	(-, E1.val ,T.val, t2)
E → T	E.val := T.val	
T → T * F	T.val := T1.val * F.val	(*, T1.val, F.val, t3)
T → F	T.val := F.val	
F → (E)	F.val := E.val	
F → nb	F.val := nb.val	

On va traduire l'opération **(3 + 2) * 4** : **(+,3,2,t1) t1=5**

(*,t1,4,t2) t2=20

Chapitre 8. Contrôle de type

8.1 Introduction :

Le **contrôle de type** dans les compilateurs est un mécanisme essentiel utilisé pour garantir que les types de données manipulés dans un programme respectent les règles et contraintes définies par le langage de programmation. Ce contrôle intervient pendant les phases de compilation et peut être statique (avant l'exécution) ou dynamique (pendant l'exécution).
Les objectifs de contrôle de type sont :

- **Prévenir les erreurs d'exécution** : En détectant des incohérences avant que le programme ne s'exécute.
- **Assurer la robustesse du code** : En imposant des règles strictes sur les opérations permises pour chaque type.
- **Optimiser les performances** : Les types étant connus à la compilation, le compilateur peut générer un code plus efficace.
- **Améliorer la lisibilité et la maintenance du code** : Les développeurs peuvent comprendre clairement ce qu'un programme est censé faire grâce aux déclarations explicites de type.

8.2 Système de typage :

Un système de typage est une **collection de règles formelles** qui permet :

- **D'attribuer des types** aux différentes parties d'un programme (variables, expressions, fonctions, etc.).
- **De garantir la sécurité** en interdisant les opérations qui pourraient produire des comportements indéfinis ou des erreurs à l'exécution.

8.2.1 Expressions de types :

Une **expression de type** est une construction syntaxique utilisée pour décrire un type dans un langage de programmation. Elle permet de représenter à la fois les **types simples** (comme les types de base) et les **types composés** formés en combinant ou en paramétrant d'autres types.
De manière générale, une **expression de type** peut être l'une des suivantes :

8.2.1.1 Expressions de types : types de base :

Est un type fondamental fourni par le langage de programmation, comme :

- **booléen** : Bool (ex. pour des valeurs logiques true/false).
- **entier** : Int (ex. pour des nombres entiers comme 42).
- **réel** : Float ou Double (ex. pour des nombres décimaux comme 3.14).

- **caractère** : Char (ex. pour un seul caractère comme 'A').
- **type vide** : Void ou un équivalent, utilisé pour représenter l'absence de valeur.
- **erreur_de_type (type_error)** : type spécial utilisé par le compilateur pour signaler et propager des erreurs de types dans les expressions.

Expressions de types : noms de types

8.2.1.2 Expressions de types : noms de types :

Un est type défini à partir d'une autre expression de type :

Exemple : type Age = Int (ici, Age est une expression de type basée sur Int).

8.2.2 Construction de types :

La construction de types est un processus qui permet de définir des types complexes ou composés à partir de types de base, en utilisant des constructeurs de types. Cela permet de modéliser des structures de données, des fonctions et des comportements complexes dans un langage de programmation.

8.2.2.1 Constructeurs de types : le tableau :

Un **tableau** est une collection ordonnée d'éléments de même type.

Notation : array(I , T) :

- I : Indice (ensemble d'indices comme [1,N]).
- T : Type des éléments.

Exemple :

Array ([1,10],int) : Un tableau d'entiers indexé de 1 à 10.

8.2.2.2 Constructeurs de types : le produit Cartésiens :

Un **produit cartésien** représente une paire (ou un tuple) de types.

Notation : T1×T2 : Combinaison de deux types.

Exemple :

int×float : Une paire contenant un entier et un nombre flottant.

8.2.2.3 Constructeurs de types : les structures :

Une **structure** regroupe plusieurs champs nommés, chacun ayant son propre type.

Notation : {nom1:T1,nom2:T2,… }

Exemple :

Une structure représentant une personne : Personne={nom: String, âge :int}

8.2.2.4 Constructeurs de types : les pointeurs :

Un pointeur est une référence vers une zone mémoire contenant une valeur.

Notation : *T ou Pointer[T]

Exemple :

*int : Un pointeur vers un entier.

8.2.2.5 Constructeurs de types : les fonctions :

Un **type fonctionnel** associe un domaine D à un co-domaine A.

Notation: D→A

- D : Type des entrées.
- A : Type des sorties.

Exemple :

int→bool : Une fonction qui prend un entier et retourne un booléen.

8.2.3 Construction par Composition

Les constructeurs de types peuvent être combinés pour former des structures plus complexes.

Exemples :

Tableau de paires

Array ([1,N],(int×float)) : Représente un tableau de N paires, où chaque paire contient un entier et un flottant.

Structure imbriquée

{nom : String, coordonnées : (int×int)} : Représente une structure avec un champ nom de type chaîne de caractères et un champ coordonnées de type paire d'entiers.

8.2.4 Formalisme des Constructions de Types :

Formellement, on peut définir une expression de type T comme suit :

- **Types de base** :

T → Bool | Int | Float | Char | Void

- **Constructeurs de types** :
  T → array (I , T) | T1×T2 | {nomi : Ti} | T^ | D→A

8.3 Contrôle de type statique et dynamique :

Le **contrôle de type** est une étape clé dans la gestion des erreurs liées aux types de données dans un programme. Il peut être effectué de deux manières principales : **statiquement** (avant l'exécution) ou **dynamiquement** (pendant l'exécution).

8.3.1 Contrôle de Type Statique :

Le contrôle de type statique est réalisé par le **compilateur**, avant l'exécution du programme. Il a les caractéristiques suivantes :

- Les types de toutes les expressions et variables sont vérifiés **avant** que le programme ne soit exécuté.
- Les erreurs de type sont signalées comme des erreurs de compilation.
- Garantit que, si le programme compile correctement, il est exempt d'erreurs de type (dans les limites des vérifications statiques).

Avantages :

- **Performance accrue** : Pas de surcoût lié à la vérification des types pendant l'exécution.
- **Détection précoce des erreurs** : Les erreurs sont détectées au moment de la compilation.
- **Sécurité** : Les programmes typés statiquement offrent une plus grande assurance sur leur comportement.

Inconvénients

- **Manque de Flexibilité :** Difficile de travailler avec des types dynamiques ou inconnus.
- **Code Plus Long :** Nécessite de déclarer les types, ce qui alourdit le code.
- **Complexité :** Certains concepts sont difficiles à comprendre pour les débutants.

Exemples de langages avec typage statique :

- **Java** : Une variable déclarée comme **int** ne peut pas être affectée avec une valeur de type **String**.
- **C++** : Le compilateur vérifie les types pour détecter les incompatibilités.

8.3.2 Contrôle de Type Dynamique :

Le contrôle de type dynamique est réalisé par le **programme cible** pendant l'exécution. Il a les caractéristiques suivantes :

- Les types des valeurs sont vérifiés **au moment où elles sont utilisées**.
- Le programme peut planter avec une erreur de type si une valeur est utilisée de manière incompatible avec son type attendu.
- Nécessite que chaque valeur transporte avec elle une **méta-information** sur son type.

Avantages :

- **Flexibilité** : Permet d'écrire des programmes génériques et adaptatifs où les types sont résolus à l'exécution.
- **Simplicité pour le programmeur** : Pas besoin de déclarer explicitement les types dans certains cas.

Inconvénients :

- **Performance réduite** : Les vérifications des types à l'exécution augmentent le coût en temps d'exécution.
- **Détection tardive des erreurs** : Les erreurs de type ne sont découvertes qu'au moment où elles se produisent.

Exemples de langages avec typage dynamique :

- **Python** : Une variable peut changer de type à tout moment (x = 5 puis x = "texte").
- **JavaScript** : Très flexible, mais les erreurs de type sont courantes.
- **Ruby** : Vérifie les types uniquement à l'exécution.

8.3.3 Systèmes de Typage et Systèmes Sains :

Un **système de typage sain** est un système qui garantit qu'aucune **erreur de type** ne se produira à l'exécution si le programme est bien typé selon ses règles. Il a les caractéristiques suivantes :

- **Exclusion des contrôles dynamiques** : Toutes les erreurs de type sont détectées statiquement.
- **Fiabilité accrue** : Assure que les programmes exécutés ne planteront pas pour des raisons de type.
- **Respect du contrat des types** : Une fois le programme accepté par le compilateur, chaque opération est garantie d'être utilisée avec des valeurs du bon type.

8.3.4 Langages Fortement Typés et Langages faiblement typés

8.3.4.1 Langages Fortement Typés :

Un langage est dit **fortement typé** si son compilateur peut garantir qu'aucune erreur de type ne se produira à l'exécution.

Exemples :

- **Java** : Typage statique et vérification rigoureuse.
- **Python** : Bien qu'il soit dynamiquement typé, Python est fortement typé. Les types ne changent pas automatiquement.
- **C#** : Implique des règles strictes sur les types, avec des conversions explicites nécessaires.

8.3.4.2 Langages faiblement typés :

Les langages faiblement typés permettent des conversions implicites entre types, ce qui peut entraîner des erreurs subtiles.

Exemples :

- **C** : Permet des conversions implicites entre types sans avertissement.

- **JavaScript** : Très flexible, mais les conversions implicites peuvent causer des comportements inattendus.
- **PHP** : Les types changent dynamiquement selon le contexte.

8.4 Un système de typage simple :

Nous définissons un contrôleur de type pour un langage simple, où chaque identificateur doit obligatoirement être déclaré avec son type avant d'être utilisé. Ce contrôleur repose sur un schéma de traduction qui déduit le type de chaque expression en fonction des types de ses sous-expressions.

La grammaire et les règles sémantiques suivantes définissent le processus d'attribution d'un type lors de la déclaration des variables.

D → L id;

L → L id, | T

T → int | float | char | array[nb] of T | *T

Production	**Règle sémantique**
D → L id;	TS.inserer(id.nom , L.type) /* TS : tableau de symbole*/ D.type = L.Type;
L → L_1 id,	TS.inserer(id.nom , L_1.type) /* TS : tableau de symbole*/ L.type = L_1.type
L → T	L.type = T.type
T → int	T.type = entier
T → float	T.type = réel
T → char	T.type = caractère
T → array[nb] of T_1	T.type = tableau(nb.val, T_1.type)
T → *T	T.type = pointeur (T1.type)

La fonction **inserer(x,t)** est utilisée pour insérer le type **t** du variable **x** dans la table des symbole.

8.4.1 Contrôle de type des expressions :

La grammaire et les règles sémantiques ci-dessous permettent de contrôler et d'attribuer des types aux **expressions arithmétiques**. Cette approche repose sur une traduction dirigée par la syntaxe, de type S-attribuée, où tous les attributs sont synthétisés. Une fonction

rechercher_type(e) est utilisée pour récupérer le type associé à une entrée e dans la table des symboles.

E → litteral

E → nb

E → id

E → E mod E

E → id [E]

E → *E

Production	**Règle sémantique**
E → litteral	E.type = caractère
E → nb	E.type = entier
E → id	E.type = TS.rechercher_type(id.nom)
E → E_1 mod E_2	**Si** E_1.type == E_2.type == entier **alors** E.type = entier **Sinon** E.type = erreur_de_type
E → id [E_1]	**Si** E_1.type == s et id.type == tableau(s,t) **alors** E.type = t **Sinon** E.type = erreur_de_type
E → *E_1	**Si** E_1.type == pointeur(t) **alors** E.type = t **Sinon** E.type = erreur_de_type

8.4.2 Contrôle de type des instructions :

Les instructions n'ayant pas de valeur associée se voient attribuer un type de base vide. La grammaire et les règles sémantiques ci-dessous permettent de contrôler les instructions telles que l'affectation, les structures conditionnelles et les boucles. Cette approche repose sur les règles précédentes pour le contrôle des expressions, car les instructions peuvent contenir des expressions.

I → id = E

I → if E then I

I → while E do I

I → I;I

Production	Règle sémantique
I → id = E	**Si** id.type == E.type **alors** I.type = vide **Sinon** I.type = erreur_de_type
I → if E then I_1	**Si** E.type == booléen **alors** I.type= I_1.type **Sinon** I.type = erreur_de_type
I → while E do I_1	**Si** E.type==booléen **alors** I.type= I_1.type **Sinon** I.type = erreur_de_type
I → I_1 ; I_2	**Si** I_1.type==I_2.type==vide **alors** I.type= vide **Sinon** I.type = erreur_de_type

8.4.3 Contrôle de type des appels de fonctions :

La grammaire et les règles sémantiques suivantes permettent de vérifier l'application d'une fonction à un argument. Lorsqu'il y a plusieurs arguments, leur type est supposé être le produit cartésien des types (c'est-à-dire une concaténation des types). Par exemple, pour **n** arguments de types **T1, ..., Tn**, on les considère comme un seul argument de type **T1 × ... × Tn**.

Remarque : l'attribut id.type est déterminé en recherchant le type de l'identificateur dans la table des symboles (TS) à l'aide de la règle suivante : **id.type = TS.rechercher_type(id.nom)**

E → id (E)

Production	Règle sémantique
E → id (E_1)	**Si** E_1.type == s et id.type == s → t alors E.type = t **Sinon** E.type = erreur_de_type

8.5 Equivalence des expressions de type :

L'algorithme de typage nécessite de nombreux tests pour évaluer l'équivalence entre types, ce qui demande une définition précise des conditions dans lesquelles deux expressions de type peuvent être considérées comme équivalentes. En théorie, deux types sont équivalents s'ils peuvent être substitués l'un par l'autre sans altérer le comportement ou le résultat du programme. Cependant, cette définition idéale n'est pas toujours respectée en pratique. De nombreux compilateurs se basent sur des notions comme l'équivalence structurelle ou l'équivalence par nom, mais ils ne parviennent pas toujours à déterminer avec exactitude si deux types sont réellement équivalents.

8.5.1 Équivalence structurelle :

Dans le contexte de l'équivalence structurelle des expressions de type, deux expressions sont considérées comme équivalentes si elles appartiennent au même type de base ou si elles sont

construites à l'aide du même constructeur appliqué à des types qui sont eux-mêmes structurellement équivalents. L'algorithme suivant permet de vérifier cette équivalence structurelle :

Fonction EquivStructurelle **(**s,d**) : booleen ;**
Début
Si s et **d** sont le même type de base **alors**
Retourner (vrai) ;
Sinon
Si s=tableau(s1,s2) et **d= tableau(d1,d2) alors**
Retourner (EquivStructurelle **(s1 ,d1) et** EquivStructurelle **(s2, d2)) ;**
Sinon
Si s= s1 X s2 et **d= d1 X d2 alors**
Retourner (EquivStructurelle **(s1, d1) et** EquivStructurelle **(s2, d2)) ;**
Sinon
Si s=pointeur (s1) et **d = pointeur(d1) alors**
Retourner (EquivStructurelle **(s1, d1)) ;**
Sinon
Si s= s1→s2 et **d= d1→d2 alors**
Retourner (EquivStructurelle **(s1,d1) et** EquivStructurelle **(s2, d2));**
Sinon
Retourner (faux) ;
Fin sinon
Fin sinon
Fin sinon
Fin sinon
Fin

Remarque : Bien que l'algorithme soit simple, le problème de l'équivalence structurelle est, dans la pratique, plus complexe dans les cas généraux en raison de plusieurs défis :

- Les expressions de type peuvent inclure des noms de types qui ne sont pas des types de base, rendant leur résolution plus difficile.

- L'imbrication des types, comme dans les structures ou les pointeurs, peut compliquer considérablement la comparaison des types, nécessitant une analyse approfondie pour évaluer leur équivalence.

8.5.2 Équivalence de noms :

Deux types sont considérés comme équivalents s'ils portent le même nom. Ce nom peut être défini soit par le programmeur lors de la déclaration, soit généré automatiquement par le compilateur.

Exemple :

Voici la déclaration suivante :

type t1**= tableau [int]** de **int;**

type t2**= tableau [int]** de **int;**

t1 et **t2** ne **sont pas équivalents** par nom.

Voici la déclaration suivante :

type t3**= tableau [int]** de **int;**

type t4**=** t3**;**

Dans ce cas **t3** et **t4 sont équivalents** par nom.

8.6 Conversions de type :

La conversion de types intervient pour plusieurs raisons :

- Les compilateurs cherchent à produire du code capable de traiter des calculs impliquant des données de types différents.
- En cas d'échec dans la vérification de l'équivalence des types, le compilateur tente de résoudre cette incompatibilité afin de poursuivre l'analyse du programme sans interruption.

Prenons l'exemple de l'expression x + i, où x est de type réel et i est de type **entier**. Étant donné que les entiers et les réels sont représentés différemment dans un calculateur et que leurs

opérations nécessitent des instructions machine distinctes, le compilateur doit souvent convertir l'un des opérandes de l'opération + afin d'uniformiser leurs types.

Cette harmonisation des types peut s'effectuer de deux manières : par une **conversion implicite**, réalisée automatiquement par le compilateur, ou par une **conversion explicite**, spécifiée directement par le programmeur.

8.6.1 Conversion implicite :

La **conversion implicite** de types, également appelée **coercition**, est définie par des règles spécifiques au langage de programmation. Par exemple, dans une opération comme **réel + entier**, le résultat sera automatiquement converti en type **réel**. Ces conversions suivent généralement le principe fondamental de préserver l'information, ce qui signifie qu'une donnée stockée sur **n bits** ne doit pas être transformée en une représentation sur **m < n bits**.

8.6.2 Conversion explicite :

Une **conversion explicite de type** consiste à appliquer une fonction ou une opération qui transforme une donnée d'un type en une autre, par exemple, convertir un entier en un réel. Cette transformation respecte les règles définies par le langage.

Remarque :

Il est essentiel de distinguer entre une conversion explicite de type et un forçage de type :

- Une **conversion explicite de type** modifie la donnée en appliquant une transformation, conformément aux normes du langage.
- Un **forçage de type**, en revanche, ne modifie pas la donnée elle-même. Il change uniquement la manière dont le compilateur interprète son type.

Le **forçage explicite de type** est une fonctionnalité disponible dans certains langages de programmation, permettant au programmeur d'indiquer directement le type d'une expression. Par exemple, en langage **C**, on utilise le **cast** sous la forme **(type) expr**. Ce mécanisme est conçu pour pallier les limites du système de typage, mais il est souvent utilisé de manière excessive, ce qui peut engendrer des problèmes. Un exemple courant est le **rétrécissement des données**, comme la conversion d'un **double en int**, qui peut entraîner des pertes d'information et rendre certains bugs difficiles à détecter.

8.7 Surcharge des opérateurs :

Un **symbole surchargé** est un symbole qui peut avoir des significations différentes en fonction du contexte d'utilisation.

Exemple mathématique :

L'opérateur **+** est surchargé en mathématiques :

- Dans l'expression **A+B**, le rôle de + varie selon que **A** et **B** sont des entiers, des nombres réels, des nombres complexes ou des matrices.

Exemple en langage Ada :

En Ada, les **parenthèses ()** sont également surchargées.

Par exemple, dans l'expression **A(I)**, les parenthèses peuvent représenter :

- L'accès à un élément d'un tableau,
- Un appel de fonction avec l'argument I,
- Ou encore une conversion explicite de I vers le type A.

Dans les langages de programmation :

Les opérateurs arithmétiques (comme +, -, *, /) sont souvent surchargés. Cependant, cette surcharge est résolue en fonction des types des arguments. Par exemple, pour l'opérateur +, les types des deux opérandes déterminent le comportement :

- Si les arguments sont des entiers, l'addition entière est effectuée.
- Si ce sont des chaînes de caractères (dans certains langages comme Python ou JavaScript), une concaténation peut avoir lieu.

La résolution de la surcharge suit une analyse par cas, semblable à celle d'une règle sémantique associée à une expression du type **E→ E1 op E2**, où le type de **E** est déterminé en fonction des types possibles de **E1** et **E2**.

Ce phénomène de surcharge est un outil puissant, permettant une meilleure expressivité et flexibilité dans les langages de programmation.

Chapitre 9. Environnement d'exécution

9.1 Introduction :

L'environnement d'exécution (runtime) joue un rôle essentiel dans la mise en œuvre d'un langage de programmation en permettant l'exécution des programmes développés avec ce langage. Il propose une gamme de services, notamment la gestion des entrées-sorties, l'arrêt des processus, l'accès aux services du système d'exploitation, le traitement des erreurs, la gestion des événements, l'interopérabilité avec d'autres langages, ainsi que des outils comme le débogage, le profilage et la collecte automatique des ressources inutilisées (garbage collection).
Dans les **langages interprétés**, le runtime fonctionne comme un interpréteur : il traite le code source en temps réel, manipule les variables, gère l'allocation de la mémoire et prend en charge la gestion des erreurs d'exécution.
Dans les **langages compilés**, les frontières du runtime sont moins clairement définies. Même si le code est directement exécuté par le processeur, certaines fonctionnalités du runtime restent indispensables, telles que la création d'objets, la vérification des types ou la gestion automatique de la mémoire, particulièrement dans les langages orientés objet.

9.2 Procédures et activations :

9.2.1 Procédures :

Une procédure est un élément fondamental en programmation qui encapsule une suite d'instructions regroupées sous un nom unique, permettant leur réutilisation et facilitant l'organisation du code. Voici un approfondissement des concepts de procédures et leurs activations :

9.2.1.1 Définition statique :

Une procédure est définie dans le code source avec un **identifiant** (nom) et un **corps**. Le corps contient une séquence d'instructions qui sont exécutées lorsqu'on appelle la procédure.

9.2.1.2 Structure générale :

La structure générale d'une procédure dans un langage de programmation est :

```
Procedure NomProcédure (parametres) ;
  Begin
    Instruction1 ;
    Instruction2 ;
    ….
    Instruction n ;
```

End ;

9.2.1.3 Rôles des procédures :

- Réduire la redondance en regroupant du code réutilisable.
- Améliorer la lisibilité et la modularité.
- Faciliter le débogage et la maintenance.

9.2.1.4 Types :

- **Procédures sans paramètres** : Aucune donnée n'est passée lors de l'appel.
- **Procédures avec paramètres** : Elles acceptent des arguments pour personnaliser leur exécution.
- **Procédures avec valeur de retour** : On les appelle souvent "fonctions" dans certains langages. Dans ce cas la structure générale de la procédure devient :

  Function NomFonction (**parametres**) : **Type de retour** ;

  Begin

  Instruction1 ;

  Instruction2 ;

  ….

  Instruction n ;

  NomFonction := Valeur de retour ; **/*return** Valeur de retour ; ***/**

  End ;

9.2.2 Activations :

9.2.2.1 Définition dynamique :

Une activation d'une procédure correspond à un **appel effectif** de cette procédure au cours de l'exécution.

Chaque appel d'une procédure peut être considéré comme une instance distincte de cette procédure en mémoire.

9.2.2.2 Caractéristiques d'une activation :

Chaque activation dispose d'un espace mémoire propre pour :

- Les **paramètres passés**.
- Les **variables locales**.
- Les données de retour ou d'état.

Une procédure peut être appelée plusieurs fois, chaque fois créant une **nouvelle activation**.

9.2.2.3 Pile d'activations :

Lorsqu'une procédure est appelée, une **entrée (frame)** est ajoutée à la pile d'appels (call stack) pour enregistrer les données nécessaires à cette activation.

Une fois la procédure terminée, cette entrée est retirée de la pile.

Ce mécanisme est essentiel pour gérer les appels imbriqués et la récursivité.

9.2.2.4 Arbre d'activation :

L'**arbre d'activation** est un modèle utilisé pour représenter les relations entre les activations de procédures au cours de l'exécution d'un programme. Il est particulièrement utile pour comprendre l'ordre des appels, leur hiérarchie, et leur relation dans le temps, notamment dans des contextes où des procédures peuvent s'appeler récursivement ou mutuellement.

Exemple :

Dans cet exemple nous avons définie deux procédures et une fonction comme suit :

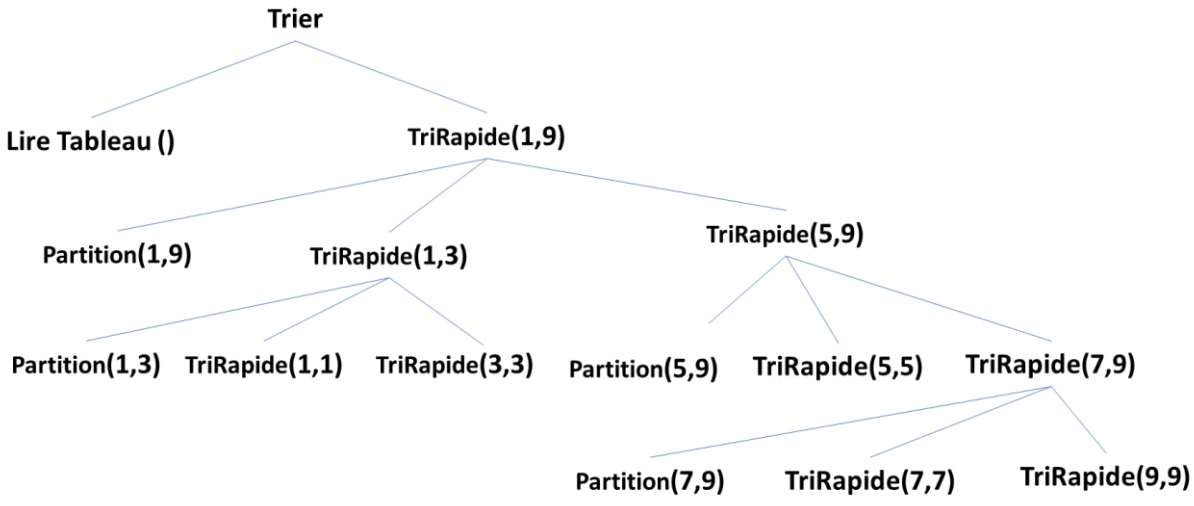
```
procedure LireTableau () ;
    var i : integer;
begin
  for i:=1 to 9 do
     read(t[i]);
end;

function Partition(y,z: integer) : integer;
 var  i, j, k, x : integer;
      u : array[y,z] of integer;
 begin
   for i := y to z do
       u[i] := t[i];
   x := t[y]; j := y; k := z;
   for i := y+1 to z do
      if u[i] < x then
         begin
           t[j]:=u[i];
           j:=j+1;
         end
          else
```

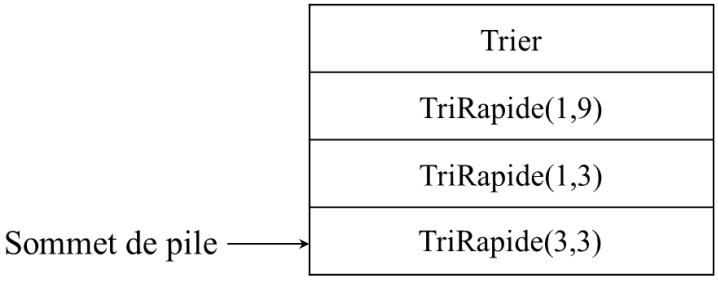
```
                begin
                  t[k] :=u[i];
                  k:=k-1;
                end;
        Partition := j;
  end;

procedure TrieRapide(m,n:integer);
        var i : integer;
  begin
    if n > m then
       begin
          i:=Partition(m,n);
          TriRapide(m,i-1);
          TriRapide(i+1,n);
       end;
end;
Program Trier (input,output);
   var t  :array[1..9] of integer;
    begin
        LireTableau();
        TriRapide(1,9);
     end.
```

Voici l'arbre d'activation pour cet exemple :



Voici une partie de la pile d'activations :

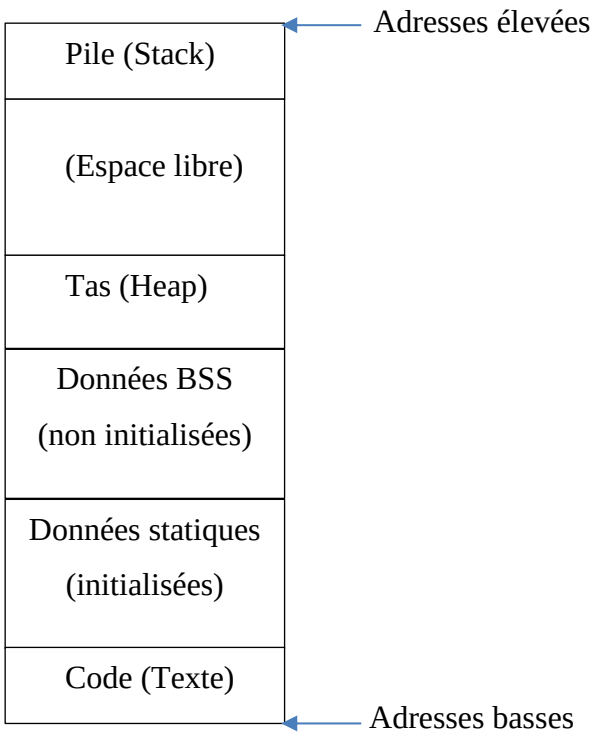


9.3 Organisation de l'espace mémoire :

L'**organisation de l'espace mémoire** est un aspect essentiel de l'exécution d'un programme. Elle définit comment les données (code, variables, objets, etc.) sont disposées en mémoire pour permettre un accès efficace et sécurisé pendant l'exécution. Cette organisation repose sur plusieurs segments logiques, chacun ayant un rôle précis.

9.3.1 Principaux segments de la mémoire :

Un programme est généralement organisé en plusieurs zones mémoire. Voici une représentation classique de l'organisation en mémoire d'un programme (du bas vers le haut de l'espace mémoire) :



9.3.1.1 Segment de code (ou texte) :

Contient les instructions exécutables du programme. Ce segment est généralement en lecture seule pour empêcher des modifications accidentelles ou malveillantes.

Exemple :

Les instructions de la fonction **main()** ou les bibliothèques liées statiquement.

9.3.1.2 Segment de données statiques (ou initialisées) :

Stocke les **variables globales** et les **données statiques** initialisées. Les variables ici conservent leur valeur tout au long de l'exécution.

Exemple :

int x = 5; déclaré en dehors de toute fonction.

9.3.1.3 Segment de données non initialisées (BSS - Block Started by Symbol) :

Contient les variables globales ou statiques non initialisées (elles sont initialisées par défaut à zéro).

Exemple :

static int compteur ;

9.3.1.4 Tas (Heap) :

Zone dédiée à l'**allocation dynamique de mémoire**. La mémoire est allouée et libérée manuellement par le programmeur ou automatiquement par un mécanisme de gestion (ex. Garbage collection). Croît généralement vers le haut de la mémoire.

Exemple :

Utilisation de **malloc()** en **C** ou **new** en **JAVA** et **C++**.

9.3.1.5 Pile (Stack) :

Utilisée pour stocker les **variables locales**, les **paramètres de fonction**, et les **adresses de retour**. Chaque appel de fonction crée un **frame d'activation** dans la pile. La pile suit une structure **LIFO** (Last In, First Out). Croît généralement vers le bas de la mémoire.

Exemple :

int a = 10; déclaré dans une fonction.

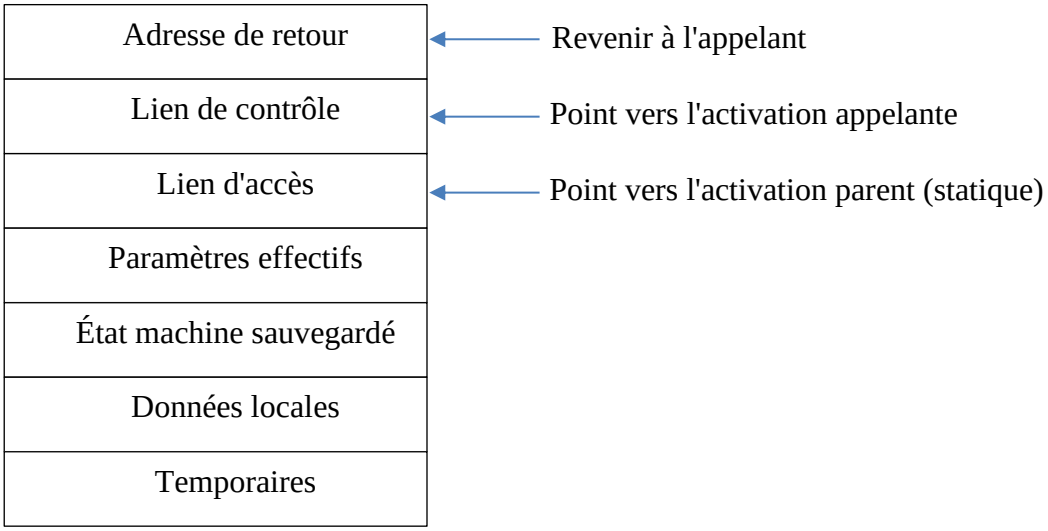
9.4 Bloc d'activation :

Un **bloc d'activation** (ou **record d'activation**) est une structure en mémoire, généralement située dans la **pile d'exécution**, qui contient toutes les informations nécessaires pour gérer une **activation** d'une procédure ou d'une fonction.

Il est créé lorsqu'une procédure est appelée et détruit lorsqu'elle se termine. Ce bloc est essentiel pour gérer l'ordre des appels, la récursivité et la gestion des variables locales et des paramètres.

9.4.1Composantes d'un bloc d'activation :

La structure suivante est une représentation typique des composantes d'un bloc d'activation, bien qu'elle puisse varier selon le langage et le compilateur :



9.4.1.1 Adresse de retour :

Contient l'adresse où le contrôle doit revenir après l'exécution de la procédure ou fonction appelée. Elle est essentielle pour gérer la pile d'appels.

Exemple : Lorsque la procédure **A()** appelle **B()**, l'adresse de retour indique où reprendre dans **A()** après l'exécution de **B()**.

9.4.1.2 Paramètres effectifs :

Ce sont les valeurs ou les références passées en argument à la procédure lors de son appel. Ils sont stockés dans le bloc d'activation pour être accessibles pendant l'exécution de la procédure.

9.4.1.3 Lien de contrôle (ou lien dynamique) :

Le lien de contrôle indique le bloc d'activation du programme appelant (la fonction ou procédure qui a initié l'appel). Il permet de revenir au contexte de l'appelant après l'exécution. Il est utile pour suivre la séquence des appels imbriqués.

9.4.1.4 Lien d'accès (ou lien statique) :

Il est utilisé pour accéder aux variables non locales dans des langages supportant des portées imbriquées (par exemple, dans des langages comme Pascal). Ce lien pointe vers le bloc d'activation de la portée parent (statique) de la fonction en cours.

9.4.1.5 État machine sauvegardé :

Il contient les informations nécessaires pour restaurer l'état du programme après l'exécution de la procédure. Il inclut des registres importants (par exemple, le compteur de programme, pointeurs de pile).

9.4.1.6 Données locales :

Espace réservé pour les **variables locales** de la procédure. Ces variables sont créées à chaque activation et détruites à la fin de la procédure.

9.4.1.7 Temporaires :

C'est un espace utilisé pour stocker des valeurs intermédiaires pendant l'exécution d'une procédure. Il est utile pour évaluer des expressions complexes ou stocker des résultats avant leur transfert.

9.4.2 Relation avec la pile d'exécution

Chaque **appel de fonction ou procédure** ajoute un nouveau bloc d'activation sur la pile. La pile suit une structure **LIFO** :

- Une fonction appelle une autre : son bloc est empilé.
- À la fin de l'exécution, son bloc est dépilé, et le contrôle revient au programme appelant.

9.5 Allocation de la mémoire :

L'**allocation de la mémoire** consiste à réserver des espaces en mémoire pour stocker les données nécessaires à l'exécution d'un programme. La gestion de la mémoire est essentielle pour garantir l'efficacité, la stabilité et la sécurité des logiciels. Elle peut être statique ou dynamique, selon le moment où l'espace mémoire est attribué. Il existe plusieurs types d'allocation mémoire :

9.5.1 Allocation statique :

La mémoire est allouée au moment de la compilation, et l'espace réservé reste constant pendant toute la durée d'exécution du programme.

Caractéristiques :

- Les variables globales et statiques utilisent ce type d'allocation.

- L'espace mémoire est réservé avant l'exécution.
- Moins flexible mais rapide et déterministe.

Exemple :

```
int x = 10;  // Allocation statique pour une variable globale
static int y;  // Allocation statique pour une variable locale statique
```

9.5.2 Allocation dynamique sur le tas (heap):

La mémoire est allouée pendant l'exécution du programme, en fonction des besoins. Cette mémoire est généralement gérée sur le **tas** (**heap**).

Caractéristiques :

- Permet une grande flexibilité et une gestion efficace des ressources pour des structures dont la taille est inconnue à la compilation.
- L'espace peut être libéré lorsque les données ne sont plus nécessaires.
- Risque d'erreurs comme les **fuites mémoire** ou les **débordements**.

Exemple :

```
int* ptr = (int*) malloc(sizeof(int));  // Allocation dynamique
*ptr = 42;
free(ptr);  // Libération de la mémoire
```

9.5.3 Allocation dynamique sur la pile (stack) :

La mémoire est allouée automatiquement pour les variables locales d'une fonction ou d'une procédure. L'espace est libéré lorsque la fonction termine.

Caractéristiques :

- Utilise la **pile** (stack).
- Rapide, mais limitée en taille.

Exemple :

```
void fonction() {
    int a = 5;  // Allocation automatique (pile)
   }
```

9.6 Accès au noms non locaux :

L'**accès aux noms non locaux** concerne la manière dont un programme accède aux variables définies dans un **contexte extérieur** à la fonction ou au bloc de code en cours d'exécution. Ce concept est particulièrement pertinent dans les langages qui supportent les **portées lexicales** (ou **statistiques**) et les structures imbriquées, comme Pascal, C, ou Python.

Un **nom non local** désigne une variable qui :

- N'est pas définie dans le **bloc courant** (la fonction ou procédure en cours d'exécution).
- Est définie dans un **bloc englobant** (portée extérieure mais visible).

Exemple :

```
int x = 10;  // Nom global (non local dans les fonctions)
void f() {
   int y = 20;  // Nom local à f
   void g() {
      int z = 30;  // Nom local à g
      printf("%d", x);  // Accès à un nom non local (x)
      printf("%d", y);  // Accès à un nom non local (y)
   }
   g();
}
```

Dans cet exemple :

- x est global et accessible dans g() car c'est un nom non local.
- y est local à f(), mais pour g(), il s'agit également d'un nom non local.

9.6.1 Portée lexicale et règles d'accès :

9.6.1.1 Portée lexicale (ou statique) :

La portée d'une variable est déterminée **au moment de la compilation**, en fonction de la structure du code. Les noms non locaux sont cherchés dans les blocs englobants dans l'ordre hiérarchique, jusqu'à la portée globale.

9.6.1.2 Règles de recherche :

Pour chercher un Identifiant (variable, fonction, procédure, constante, symbole défini) on suit les étapes suivantes :

1. Chercher dans la portée locale (bloc courant).
2. Si le nom n'est pas trouvé, chercher dans la portée immédiatement supérieure (bloc englobant).
3. Répéter jusqu'à atteindre la portée globale.
4. Si le nom reste introuvable, une erreur est signalée (variable non définie).

9.6.2 Gestion des noms non locaux :

9.6.2.1 Lien d'accès (Static Link) :

Implémentation classique dans les langages comme Pascal. Chaque **bloc d'activation** contient un **lien statique** pointant vers le bloc d'activation de son **parent lexical**. Ce lien permet d'accéder aux variables des portées englobantes.

Exemple (en Pascal) :

```
procedure A;
  var x: integer;
        procedure B;
          begin
           x := x + 1;  // Accès au x défini dans A via le lien statique
          end;
 begin
   B;
 end;
```

Dans ce cas :

- Lorsque B est appelé, le lien statique de son bloc d'activation pointe vers celui de A.
- x dans B est résolu grâce au lien statique.

9.6.2.2 Table des symboles :

Les compilateurs créent une **table des symboles** pour gérer les noms et leurs portées. Lorsqu'une variable non locale est utilisée, elle est cherchée dans la table des symboles pour retrouver sa définition.

9.6.2.3 Chaîne de blocs d'activation :

Les liens statiques forment une **chaîne** traversée lors de l'accès à un nom non local. Le temps d'accès peut dépendre de la profondeur de la chaîne (mais reste efficace pour la plupart des programmes).

9.6.3 Noms non locaux dans différents langages :

a) Java

- Les noms non locaux en Java incluent :
  - Variables d'instance et de classe.
  - Variables locales capturées dans des lambdas ou des classes internes.
- Java utilise la portée lexicale pour résoudre les noms.
- Les variables globales n'existent pas strictement, mais les variables statique peuvent jouer un rôle similaire.

b) Pascal

- Les fonctions imbriquées peuvent accéder aux variables des portées englobantes.
- Utilise des liens statiques pour gérer les noms non locaux.

c) C

- Les fonctions ne sont pas imbriquées, donc l'accès aux variables des portées englobantes est limité aux variables globales.
- Les variables globales sont accessibles depuis n'importe quelle fonction.

d) Python

- Permet d'accéder aux variables définies dans des portées extérieures via le mot-clé `nonlocal`.

Exemple :

```
def outer():
    x = 10
    def inner():
        nonlocal x
        x += 1
    inner()
    print(x)  # Affiche 11
```

e) JavaScript

- Utilise des **fermetures** (closures) pour capturer les variables des portées extérieures.

Exemple :

```
function outer() {
    let x = 10;
    function inner() {
        console.log(x);  // Accès à x, non local
    }
    inner();
}
```

9.6.4 Portée dynamique (dans certains langages) :

Contrairement à la portée statique, la portée dynamique résout les noms en fonction de la chaîne des appels au moment de l'exécution. Moins courante, utilisée dans des langages comme **Lisp** ou certains interpréteurs Shell.

9.7 Passage de paramètres :

Le passage de paramètres est la manière dont les arguments sont transmis à une fonction, procédure ou méthode lors de son appel. Les mécanismes varient selon les langages et peuvent inclure des différences entre le passage par valeur, par référence, ou encore des approches spécifiques comme en Java ou Python.

9.7.1 modes de passage de paramètres :

Il existe plusieurs modes de passage de paramètres :

9.7.1.1 Passage par valeur :

Une copie de la valeur de l'argument est transmise à la fonction.

Les modifications faites au paramètre dans la fonction n'affectent pas la variable d'origine.

9.7.1.2 Passage par référence :

Une référence (adresse mémoire) à la variable d'origine est transmise.

Les modifications faites au paramètre dans la fonction affectent directement la variable d'origine.

9.7.1.3 Passage par valeur-résultat :

Une copie de la valeur est transmise à l'entrée, mais à la fin de l'exécution, la variable d'origine est mise à jour avec la valeur modifiée du paramètre.

9.7.1.4 Passage par nom :

Une expression est passée sous forme de texte ou de symbole et évaluée à chaque utilisation dans la fonction (peu commun, mais utilisé dans des langages comme ALGOL).

9.7.2 Passage de paramètres dans différents langages :

9.7.2.1 Pascal :

Modes disponibles :

- **Par valeur** (par défaut) : Les arguments sont copiés.
- **Par référence** (avec var) : Une référence à la variable d'origine est transmise.

Exemple :

```
procedure Modifier(var x: Integer);
  begin
     x := x + 10;  // Modifie la variable d'origine
  end;
var a: Integer;
 begin
   a := 5;
   Modifier(a);
   WriteLn(a);  // Affiche 15
 end;
```

9.7.2.2 C :

Modes disponibles :

- **Par valeur** (par défaut) : Les arguments sont copiés.
- Pour simuler le **passage par référence**, on utilise des **pointeurs**.

Exemple :

```
void modifier(int *x) {
   *x += 10;  // Modifie la valeur à l'adresse pointée
}
int main() {
   int a = 5;
   modifier(&a);
   printf("%d", a);  // Affiche 15
```

```
    return 0;
}
```

9.7.2.3 Python :

Passage des arguments :

- Les objets sont transmis par **référence**, mais Python traite les variables comme des références à des objets.
- Si l'objet est **mutable** (ex. : liste, dictionnaire), les modifications affectent l'original.
- Si l'objet est **immutable** (ex. : entier, chaîne), une copie de la référence est utilisée, et la modification ne change pas l'original.

Exemple :

```
def modifier(x):
    x.append(10)  # Modifie la liste originale

a = [5]
modifier(a)
print(a)  # Affiche [5, 10]
```

9.7.2.4 Java :

Java utilise uniquement le **passage par valeur**, mais la gestion des objets peut donner l'impression d'un passage par référence :

- Pour les **types primitifs** (int, float, etc.), une copie de la valeur est transmise.
- Pour les **objets**, une copie de la référence est transmise, permettant de modifier les champs de l'objet, mais pas de réassigner l'objet lui-même.

Exemple :

```
public class Main {
    public static void modifier(int x) {
        x += 10;  // Modifie uniquement la copie
    }
    public static void modifierObjet(StringBuilder sb) {
        sb.append(" Monde");  // Modifie l'objet original
    }
    public static void main(String[] args) {
        int a = 5;
        modifier(a);
        System.out.println(a);  // Affiche 5 (pas modifié)
        StringBuilder sb = new StringBuilder("Bonjour");
        modifierObjet(sb);
        System.out.println(sb);  // Affiche "Bonjour Monde"
    }
}
```

9.7.2.5 C++ :

Modes disponibles :

- **Par valeur** (par défaut) : Les arguments sont copiés.
- **Par référence** : Utilisation de & dans la signature de la fonction.
- **Par référence constante** : Utilisé pour éviter la copie tout en protégeant l'argument contre les modifications.

Exemple :

```
void modifier(int &x) {
    x += 10;  // Modifie directement la variable originale
}
int main() {
    int a = 5;
    modifier(a);
    std::cout << a;  // Affiche 15
    return 0;
}
```

Chapitre 10. Génération de code

10.1 Introduction :

La **phase de production de code** constitue la dernière étape du processus de compilation. Elle prend comme point de départ le code intermédiaire produit par les phases précédentes. Durant cette phase, un code machine spécifique est généré, adapté à l'architecture cible. Cela signifie que le code produit est dépendant de la machine cible, rendant nécessaire une adaptation pour chaque type d'architecture matérielle.

10.2 Machine cible :

Dans ce chapitre, nous nous intéressons à une machine fictive largement inspirée de l'architecture des processeurs Intel.

10.2.1 Caractéristiques de la machine :

- **Registres** :

  La machine dispose de **N registres** numérotés de $\mathbf{R_0, R_1, \ldots, R_{N-1}}$.

- **Taille des mots** :

  Chaque mot occupe **4 octets**.

- **Adressabilité** :

  La machine est **adressable par octet**, ce qui signifie que chaque octet en mémoire possède une adresse unique.

- **Format des instructions** :

  Les instructions suivent un format à **deux adresses** :

Op source destination

Signification : destination ← destination op source. Cela indique que l'opération **op** est effectuée entre les opérandes spécifiés, et le résultat est stocké dans **destination**.

Exemple :

ADD R_1 R_2 /* Le contenu de $\mathbf{R_1}$ est ajouté à $\mathbf{R_2}$, et le résultat est stocké dans $\mathbf{R_2}$. */

LOAD 100 R_1 /* Charge la valeur située à l'adresse mémoire **100** dans le registre $\mathbf{R_1}$. */

STORE R_2 200 /* Sauvegarde le contenu du registre $\mathbf{R_2}$ à l'adresse mémoire **200**. */

MOV R_1 R_2 /* Le contenu de $\mathbf{R_1}$ est copié dans $\mathbf{R_2}$. */

CMP R_2 R_3 /* Si $\mathbf{R_2} > \mathbf{R_3}$, un indicateur de **supériorité** est activé.
Si $\mathbf{R_2} < \mathbf{R_3}$, un indicateur d'**infériorité** est activé.
Si $\mathbf{R_2} = \mathbf{R_3}$, un indicateur d'**égalité** est activé. */

10.2.2 Modes d'adressage :

Les **modes d'adressage** définissent comment une instruction accède aux opérandes (données) en mémoire ou dans les registres. Chaque architecture de processeur prend en charge un ensemble spécifique de modes d'adressage. Voici une liste des principaux modes d'adressage, avec des exemples.

10.2.2.1 Adressage immédiat :

Dans ce mode, l'opérande est directement inclus dans l'instruction.

Exemple :

MOV 5 R_1 /* La valeur immédiate **5** est copiée dans le registre **R_1**. */

10.2.2.2 Adressage direct :

L'adresse mémoire où se trouve l'opérande est spécifiée directement dans l'instruction.

Exemple :

MOV [100] R_2 /* La valeur située à l'adresse mémoire **100** est copiée dans le registre **R_2**.*/

10.2.2.3 Adressage indirect :

L'adresse de l'opérande est stockée dans un registre, qui sert de pointeur.

Exemple :

MOV [R_3] R_4 /*L'adresse contenue dans **R_3** est utilisée pour accéder à la mémoire, et la valeur trouvée à cette adresse est copiée dans **R_4**. */

10.2.2.4 Adressage par registre :

L'opérande est dans un registre, et ce registre est directement spécifié.

Exemple :

MOV R_1 R_5 /* Le contenu de **R_1** est copié dans **R_5**.*/

10.2.2.5 Adressage indexé :

L'adresse effective de l'opérande est calculée en ajoutant un décalage (offset) à un registre.

Exemple :

MOV [R_3 + 4] R_6 /* Ajoute 4 à la valeur contenue dans R_3 pour obtenir une adresse mémoire. La valeur située à cette adresse est copiée dans R_6. */

10.2.2.6 Adressage relatif :

L'adresse effective est calculée en ajoutant un décalage par rapport au compteur de programme (PC). Utilisé pour des sauts ou des branchements.

Exemple :

JMP [PC + 10] /*Saute à l'adresse obtenue en ajoutant 10 à la valeur actuelle du compteur de programme. */

10.2.2.7 Adressage par base + index :

Combine un registre de base et un registre d'index pour calculer l'adresse effective.

Exemple :

MOV [R_1 + R_2] R_7 /* Additionne les valeurs de **R_1** et **R_2** pour obtenir une adresse mémoire. La valeur à cette adresse est copiée dans **R_7**. */

10.2.2.8 Adressage par pile (stack) :

Les données sont accessibles via une pile, utilisant des instructions comme PUSH ou POP.

Exemple :

PUSH R_1 /* Empile la valeur de **R_1** sur la pile */

POP R_2 /* Dépile la valeur au sommet de la pile et la stocke dans **R_2***/

10.2.2.9 Comparatif des modes :

Mode	Avantage	Inconvénient
Immédiat	Rapide et simple	Limité à des constantes
Direct	Accès facile à une adresse fixe	Peu flexible si les données changent
Indirect	Très flexible	Plus lent (accès mémoire indirect)
Par registre	Très rapide	Nombre limité de registres
Indexé	Adapté pour parcourir des tableaux	Nécessite un calcul d'adresse
Relatif	Compact pour les sauts	Limité à la portée du décalage
Base + index	Flexible pour structures complexes	Plus complexe à implémenter
Par pile	Parfait pour gérer les appels de fonctions	Moins performant pour autres usages

10.3 Blocs de base et graphes de flot de contrôle :

Les **blocs de base** et les **graphes de flot de contrôle** sont des concepts fondamentaux dans l'analyse de programmes, particulièrement dans la compilation et l'optimisation de code.

10.3.1 Blocs de base :

Un **bloc de base** est une séquence maximale d'instructions dans un programme qui :

- Est **entrée** uniquement par sa première instruction.
- Est **sortie** uniquement à la fin du bloc, sauf en cas de saut explicite.

Caractéristiques :

- Un bloc de base ne contient aucun point de branchement, sauf éventuellement à sa dernière instruction.
- Les instructions sont exécutées séquentiellement.

Exemple :

Supposons le pseudo-code suivant :

```
    a = b + c
    if a > 10 goto L1
    d = a * 2
    goto L2
L1: e = a - 1
L2: print(e)
```

Décomposition en blocs de base :

Bloc B1 :

```
a = b + c
if a > 10 goto L1
d = a * 2
goto L2
```

Bloc B2 :

L1: e = a - 1

Bloc B3 :

L2: print(e)

10.3.2 Graphes de flot de contrôle (CFG - Control Flow Graph) :

Un **graphe de flot de contrôle** est une représentation graphique d'un programme, où :

- Chaque **nœud** représente un **bloc de base**.
- Chaque **arc** représente un transfert de contrôle (un passage possible d'un bloc à un autre).

Construction :

1. Identifier tous les blocs de base.
2. Relier les blocs en fonction des transferts de contrôle.

Exemple :

Voici le graphe de flot de contrôle correspondant le pseudo-code précédent :

- **Nœuds :**

B1, B2, B3 (les blocs de base identifiés précédemment).

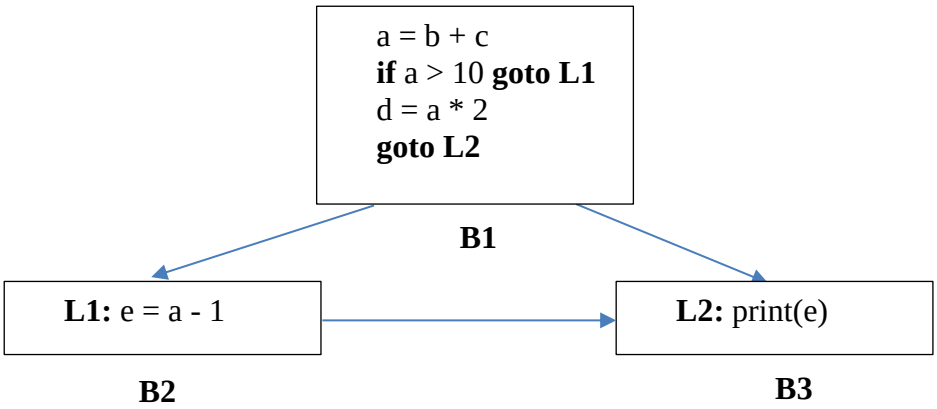
- **Arcs :**

B1→B2 : Si la condition a>10 est vraie.

B1→B3 : Si la condition a>10 est fausse (par le **goto L2**).

B2→B3 : Le programme passe à **L2** après avoir exécuté **B2**.

Représentation graphique :



10.3.3 Applications des blocs de base et Graphes de flot de contrôle (CFG) :

1. **Optimisation de code** :
   - Élimination de code mort.
   - Fusion de blocs inutiles.
   - Réduction de chemins inutiles.
2. **Analyse de portée des variables** :
   - Détection des variables vivantes (liveness analysis).
   - Détection de définitions inutilisées.
3. **Génération de code machine** :
   - Planification de registres.
   - Génération de chemins optimaux pour l'exécution.
4. **Détection des boucles** :
   - Identifier les boucles dans le graphe pour des optimisations spécifiques (par exemple, déroulage de boucle).

10.3.4 Algorithme de partitionnement :

1. **Initialiser** une **liste vide** `**tetes**` pour stocker les instructions de tête.
2. Parcourir les instructions du programme :
   a. Ajouter la première instruction à `**tetes**`.
   b. Si une instruction est **atteinte par** un **branchement**, l'ajouter à `**tetes**`.
   c. Si une instruction **suit** un **branchement**, l'ajouter à `**tetes**`.

3. **Initialiser** une **liste vide `blocs`** pour stocker les blocs de base.
4. Pour chaque instruction dans **`tetes`** :
    a. Créer un **nouveau** bloc de base (**bloc**).
    b. Ajouter les **instructions consécutives** au **bloc** jusqu'à atteindre :
        - Une autre instruction de **tête**.
        - La **fin du programme**.
    c. Ajouter le **bloc** à **`blocs`**.
5. **Retourner** la **liste** des blocs de base.

Exemple :

Reprend l'exemple précédent qui est un programme qui contient 6 instructions :

1. a = b + c
2. **if** a > 10 **goto L1**
3. d = a * 2
4. **goto L2**
5. **L1**: e = a - 1
6. **L2**: **print**(e)

Étape 1 : Identifier les instructions de tête

- La première instruction : **1**.
- Cible de branchement : **5** (label **L1**).
- Instruction qui suit un branchement : **6** (label **L2**).

Liste des instructions de tête : **tetes = {1,5,6}**

Étape 2 : Construire les blocs de base

1. **Bloc B1 : {1,2,3,4}.**
2. **Bloc B2 : {5}.**
3. **Bloc B3 : {6}.**

Étape 3:

Blocs = {B1, B2, B3}

10.3.5 Transformations sur les blocs de Base :

Dans la suite, on va présenter une description détaillée des **transformations sur les blocs de base** et des techniques **préservant la structure des programmes** pour optimiser le code.

10.3.5.1 Élimination des sous-expressions communes :

Supprimer les calculs redondants en réutilisant des résultats déjà disponibles.

Si une expression comme **a + b** est calculée plusieurs fois sans que **a** ou **b** changent, on peut calculer cette expression une seule fois et réutiliser son résultat.

Exemple :

Avant optimisation :

```
t1 = a + b
t2 = a + b
c = t1 * d
```

Après optimisation :

```
t1 = a + b
c = t1 * d
```

10.3.5.2 Élimination du code inutile :

Supprimer les instructions qui n'ont aucun effet sur le résultat final du programme.

Les instructions qui produisent des résultats inutilisés ou qui n'affectent pas l'exécution future sont supprimées.

Exemple :

Avant optimisation :

```
a = 5
b = 6
a = 7  // Redondant, car la valeur précédente de `a` est écrasée
```

Après optimisation :

```
a = 7
b = 6
```

10.3.5.3 Renommer des variables temporaires :

Éviter les conflits de noms et améliorer la lisibilité.

Attribuer des noms distincts aux variables temporaires pour éviter les collisions, souvent dans des blocs parallèles.

Exemple :

Avant optimisation :

```
t1 = a + b
t1 = c * d  // Conflit avec la première utilisation de t1
```

Après optimisation :

```
t1 = a + b
t2 = c * d  // Utilisation d'une nouvelle variable temporaire
```

10.3.5.4 Échange d'instructions adjacentes :

Réorganiser les instructions pour améliorer la performance ou permettre d'autres optimisations, tout en préservant la logique.

Les instructions indépendantes peuvent être échangées si leur ordre n'affecte pas le résultat.

Exemple :

Avant optimisation :

```
a = b + c
d = e * f
```

Après optimisation (si les deux opérations sont indépendantes) :

```
d = e * f
a = b + c
```

10.3.5.5 Transformations algébriques :

Simplifier les calculs pour réduire le coût ou le nombre d'instructions.

Utiliser des identités algébriques ou des simplifications.

Exemple :

Avant optimisation :

```
a = b * 1
c = d + 0
```

Après optimisation :

```
a = b
c = d
```

10.3.5.6 Simplification des expressions :

Réduire la complexité des expressions pour les rendre plus efficaces.

Remplacer les expressions compliquées par des équivalents plus simples.

Exemple :

Avant optimisation :

```
a = b * 2
```

Après optimisation :

```
a = b << 1  // Utilisation d'un décalage binaire pour multiplier par 2
```

10.3.5.7 Utilisation d'opérateurs moins coûteux :

Remplacer des opérations coûteuses par des opérations équivalentes mais plus rapides.

Utiliser des opérations logiques ou arithmétiques simples au lieu de divisions ou de multiplications.

Exemple :

Avant optimisation :

```
a = b / 4
```

Après optimisation :

```
a = b >> 2  // Division par 4 remplacée par un décalage binaire
```

Exemple combiné : Optimisation d'un bloc de base

Avant optimisation :

```
t1 = a + b
t2 = a + b
t3 = c * d
e = t1 + t3
f = c * d
```

Après optimisation :

```
t1 = a + b
t3 = c * d
e = t1 + t3
f = t3  // Réutilisation de t3
```

10.4 Un générateur de code simple :

Un **générateur de code simple** peut être conçu pour produire du code machine ou du code assembleur à partir d'une représentation intermédiaire comme des quadruplets.

Voici une **implémentation d'un générateur de code simple** respectant les hypothèses données. Il traduit des instructions intermédiaires en instructions machine cibles tout en gérant les registres efficacement. Les résultats des calculs sont stockés en mémoire uniquement lorsqu'ils ne peuvent plus être conservés dans les registres.

10.4.1 Hypothèses :

1. Une instruction de code intermédiaire correspond à une instruction machine cible.
2. Les registres sont utilisés pour conserver les résultats autant que possible.
3. Les contenus des registres doivent être sauvegardés avant :
    - Un appel de procédure.
    - Une instruction de branchement.
    - Une instruction étiquetée.
4. À la fin d'un bloc de base, tous les registres doivent être sauvegardés en mémoire.

Algorithme : Génération de code

Entrées :

1. Une liste d'instructions intermédiaires (quadruplets) sous forme (op,B,C,A).
2. Un nombre limité de registres disponibles (par exemple R0,R1,R2,…).

Sortie :

1. Une séquence d'instructions machine cible.

2. Un mapping entre les variables et les registres ou emplacements en mémoire.

procedure GenererCode(**liste_instructions, registres_disponibles**):
Initialiser registre_mapping (**vide**) pour suivre quelle variable est dans quel registre.
Initialiser memoire_mapping (**vide**) pour suivre l'emplacement mémoire des variables.
Initialiser code_machine (**vide**) pour stocker les instructions générées.
Début
Pour chaque instruction **(op, B, C, A)** dans liste_instructions **faire**
// Gérer l'opérande B
Si B n'est pas dans registre_mapping **alors**
Si un registre libre existe **alors**
Charger B dans un registre libre
Ajouter une instruction "**LOAD B, registre**" au code_machine
Sinon
Libérer un registre (sauvegarder son contenu en mémoire)
Charger B dans ce registre
Ajouter une instruction "**LOAD B, registre**" au code_machine
Fin Si
Fin Si
// Gérer l'opérande C
Si C n'est pas dans registre_mapping **alors**
Si un registre libre existe **alors**
Charger C dans un registre libre
Ajouter une instruction "**LOAD C, registre**" au code_machine
Sinon
Libérer un registre (sauvegarder son contenu en mémoire)
Charger C dans ce registre
Ajouter une instruction "**LOAD C, registre**" au code_machine
Fin Si
Fin Si

// Effectuer l'opération
Trouver les registres contenant **B et C**
Ajouter une instruction correspondante à "**op**" entre les deux registres au code_machine

Stocker le résultat dans le registre qui contient **C** (et **mapper A à ce registre**)

// Sauvegarde si nécessaire

Si tous les registres sont utilisés **ou** c'est la fin d'un bloc de base **alors**

Pour chaque registre contenant une variable **faire**

Sauvegarder le registre en mémoire

Ajouter une instruction "**STORE registre, memoire**" au code_machine

Mettre à jour memoire_mapping

Fin Pour

Fin Si

Fin Pour

Retourner code_machine

Fin

Exemple 1:

Dans cet exemple on va générer un code machine pour les instructions intermédiaires suivantes:

t1 = a + b ou bien (+,a,b,t1)

t2 = t1 * c ou bien (*,t1,c,t2)

d = t2 - e ou bien (-,t2,e,d)

Supposant qu'on a 3 registres : R1, R2, R3.

L'algorithme de génération de code précédent génère le code machine comme suit :

Initialisation :

- Registres disponibles : R1, R2, R3.
- registre_mapping : vide au départ.
- memoire_mapping : vide au départ.
- code_machine : vide au départ.

Instruction 1 : t1 = a + b ou bien (+,a,b,t1)

1. Vérifiez si **a** est en registre : **non**.

- Chargez **a** dans R1: **LOAD a, R1**
- Mettez à jour : **registre_mapping**[a] = R1.

2. Vérifiez si **b** est en registre : **non**.
   - Chargez **b** dans R2: **LOAD b, R2**
   - Mettez à jour : **registre_mapping**[b] = R2.
3. Effectuez l'opération R1+R2, stockez dans R2: **ADD R1, R2**
   - Mettez à jour : **registre_mapping**[t1] = R2.

Instruction 2 : t2 = t1 * c ou bien (*,t1,c,t2)

1. Vérifiez si **t1** est en registre : **oui**, **R2**.
2. Vérifiez si **c** est en registre : **non**.
   - Chargez **c** dans R3: **LOAD c, R3**
   - Mettez à jour : **registre_mapping**[c] = R3.
3. Effectuez l'opération R2×R3, stockez dans R3: **MUL R2, R3**
   - Mettez à jour : **registre_mapping**[t2] = R3.

Instruction 3 : d = t2 - e ou bien (-,t2,e,d)

1. Vérifiez si **t2** est en registre : **oui**, **R3**.
2. Vérifiez si **e** est en registre : **non**.
   - Chargez **e** dans R2: **LOAD e, R2**
   - Mettez à jour : **registre_mapping**[e] = R2.
3. Effectuez l'opération R1−R2, stockez dans R2: **SUB R3, R2**
   - Mettez à jour : **registre_mapping**[d] = R2.
4. Sauvegardez **d** en mémoire : **STORE R2, d**.

Code machine généré :

LOAD a, R1 # Charger a dans R1

LOAD b, R2 # Charger b dans R2

ADD R1, R2 # R2 = a + b = t1

LOAD c, R3 # Charger c dans R3

MUL R2, R3 # R3 = t1 * c = t2

LOAD e, R2 # Charger e dans R2

SUB R1, R3, R2 # R3 = t2 – e = d

STORE R2, d # Stocker le résultat de d en mémoire

Exemple 2 :

Dans cet exemple on va générer un code machine pour les instructions intermédiaires suivantes en utilise uniquement 2 registres :

t1 = a + b ou bien (+,a,b,t1)
t2 = t1 * c ou bien (*,t1,c,t2)
d = t2 - e ou bien (-,t2,e,d)

Étapes de génération avec R1, R2:

Initialisation :

- Registres disponibles : R1, R2.
- registre_mapping : vide.
- memoire_mapping : vide.
- code_machine : vide.

Instruction 1 : t1 = a + b ou bien (+,a,b,t1)

1. Vérifiez si **a** est en registre : **non**.
    - Chargez **a** dans R1: **LOAD a, R1**
    - Mettez à jour : **registre_mapping**[a] = R1.
2. Vérifiez si **b** est en registre : **non**.
    - Chargez b dans R2: **LOAD b, R2**
    - Mettez à jour : **registre_mapping**[b] = R2.
3. Effectuez l'opération R1+R2 stockez dans R2: **ADD R1, R2**
    - Mettez à jour : **registre_mapping**[t1] = R2.

Instruction 2 : t2 = t1 * c ou bien (*,t1,c,t2)

1. Vérifiez si **t1** est en registre : **oui**, **R2**.
2. Vérifiez si **c** est en registre : **non**.
    - Tous les registres sont occupés (R1, R2). Libérez **R1** en sauvegardant son contenu : **STORE R1, a**

- Libérez R1 pour un nouvel usage.

3. Chargez **c** dans R1: **LOAD c, R1**
   - Mettez à jour : **registre_mapping**[c] = R1.
4. Effectuez l'opération R2×R1, stockez dans R1: **MUL R2, R1**
   - Mettez à jour : **registre_mapping**[t2] = R1.

Instruction 3 : d = t2 - e ou bien (-,t2,e,d)

1. Vérifiez si **t2** est en registre : **oui**, **R1**.
2. Vérifiez si **e** est en registre : **non**.
   - Libérez **R1** en sauvegardant son contenu : **STORE R1, t1**
     - Libérez R1 pour un nouvel usage.
3. Chargez **e** dans R1: **LOAD e, R1**
   - Mettez à jour : **registre_mapping**[e] = R1.
4. Effectuez l'opération R2−R1, stockez dans R1: **SUB R2, R1**
   - Mettez à jour : **registre_mapping**[d] = R1.
5. Sauvegardez **d** en mémoire : **STORE R1, d**.

Code machine généré :

```
LOAD a, R1        # Charger a dans R1
LOAD b, R2        # Charger b dans R2
ADD R1, R2        # R2 = a + b = (t1)
STORE R1, a       # Sauvegarder R1 en mémoire pour libérer R1
LOAD c, R1        # Charger c dans R1
MUL R2, R1        # R1 = t1 * c = (t2)
STORE R1, t1      # Sauvegarder R1 en mémoire pour libérer R1
LOAD e, R1        # Charger e dans R1
SUB R2, R2, R1    # R1 = t2 - e = (d)
STORE R1, d       # Sauvegarder R1 en mémoire (d)
```

10.4.2 Avantages du générateur :

1. **Utilisation efficace des registres :** Les résultats intermédiaires sont conservés dans les registres tant que possible.
2. **Sauvegarde minimale en mémoire :** Les résultats ne sont sauvegardés qu'à la fin d'un bloc ou en cas de dépassement des registres disponibles.

3. **Génération simple et adaptable :** L'algorithme peut être ajusté pour différentes architectures de machines.

Exemple complet :

Reprend l'exemple précédent qui est un programme qui contient 6 instructions :

1. a = b + c
2. **if** a > 10 **goto L1**
3. d = a * 2
4. **goto L2**
5. **L1**: e = a - 1
6. **L2**: **print**(e)

En utilisant **l'algorithme de partitionnement** on obtient les trois blocs de base suivants :

1. **Bloc B1 : {1,2,3,4}.**
2. **Bloc B2 : {5}.**
3. **Bloc B3 : {6}.**

En nous basant sur les trois blocs issus du premier algorithme de partitionnement, on va générer le code machine en utilisant **l'algorithme de génération de code machine**.

Les registres disponibles sont : R1, R2.

Bloc 1 :

Instructions :

1. a = b + c
2. if a > 10 goto L1
3. d = a * 2
4. goto L2

Génération de code machine :

1. Charger **b** dans R1 : **LOAD b, R1**
2. Charger **c** dans R2 : **LOAD c, R2**
3. Effectuer l'opération + et stocker le résultat dans R1 : **ADD R1, R2**
4. **Mapper a à R2**.
5. Comparer **a** (dans R2) avec **10** : **CMP R2, 10**
6. Ajouter un branchement conditionnel à L1 : **JG L1**
7. Charger **2** dans R1 : **LOAD 2, R1**
8. Effectuer l'opération * pour d = a * 2 et stocker le résultat dans R1 : **MUL R2, R1**
9. **Mapper d à R1**.

10. Ajouter l'instruction pour aller à L2 : **JMP L2**.

Bloc 2 :

Instructions :

1. L1: e = a - 1

Génération de code machine :

1. Charger **1** dans R1 : **LOAD 1, R1**
2. Effectuer l'opération - pour e = a - 1 et stocker le résultat dans R3 : **SUB R2, R1**
3. **Mapper e à R1**.

Bloc 3 :

Instructions :

1. L2: print(e)

Génération de code machine :

1. Ajouter l'instruction pour imprimer **e** (dans R1) : **PRINT R1.**

Code machine complet :

```
# Bloc 1
LOAD b, R1      # Charger b dans R1
LOAD c, R2      # Charger c dans R2
ADD R1, R2      # R2 = b + c (a)
CMP R1, 10      # Comparer a avec 10
JG L1           # Si a > 10, aller à L1
LOAD 2, R1      # Charger 2 dans R1
MUL R2, R1      # R1 = a * 2 (d)
JMP L2          # Aller à L2

# Bloc 2
L1:
LOAD 1, R1      # Charger 1 dans R1
SUB R2, R1      # R2 = a - 1 (e)

# Bloc 3
L2:
PRINT R1        # Imprimer e
```

Références :

[1] : Compilateurs Principes, techniques et outils. Alfred V.Aho, Ravi Sethi, Jeffrey D.Ullman. Pearson. Paris France. 2007.

[2] : Aissa Boulmerka. Cours de Compilation. Troisième année informatique, centre universitaire de Mila. 2011 - 2012.

[3] : SAOUDI Lalia. Cours de Compilation. Troisième année informatique, Université de M'sila. 2007 - 2008.

[4] : Habib Abdulrab[1], Claude Moulin[2], Sid Touati[3]. Compilation : théorie, techniques et outils-ensemble des ressources. [1 :] Institut national des sciences appliquées de Rouen, [2 :] Université de technologie de Compiègne, [3 :] Université de Versailles Saint-Quentin en Yvelines. 2010.

[5] : Labiba Souici-Meslati. Cours de Compilation. Université d'Annaba, 2012 - 2013.

[6] : Thomas Pietrzak. Théorie des Langages Episode 6 - Analyse ascendante « Grammaires LR ». Université Paul Verlaine - Metz.

[7] : Claude Moulin. Théorie des Langages Analyse syntaxique ascendante. Université de Technologie de Compiègne Printemps 2013.

[8] : Julie Parreaux. Analyse lexicale et analyse syntaxique. Applications. 2018 - 2019

[9] : Roberto M. Amadio. Introduction à l'analyse syntaxique et à la compilation. Engineering school. Paris Diderot (Paris 7), 2009, pp.68. ffcel-00373150v2.

[10] : Éric Guérin. Grammaires et Langages. Institut national des sciences appliquées de Lyon. 2014 - 2015.

[11] : Jean Privat. Théorie et construction des compilateurs, Chapitre 3 : Evaluation des expressions régulières et automates finis. Université du Québec à Montréal. Automne 2013.

[12] : Karima Boussaha. Chapitre 02 : Les automates d'état fini. Université d'Oum El Boughi.

[13] : Olivier Bournez. Cours 8: Expressions régulières. Automates finis. LIX, Ecole Polytechnique. 2010.

[14] : Sebastien Verel. Grammaire, grammaire régulière Intelligence Artificielle et Systèmes Formels Master 1 I2L. Université du Littoral Côte d'Opale Laboratoire LISIC Equipe CAMOME. 2014 - 2015.

[15] : Didier Donsez. Outils de Compilation Lex et Yacc. Université de Grenoble Alpes. 2000. Fabrice Harrouet. Générer un analyseur avec Flex&Bison. Ecole Nationale d'Ingénieurs de Brest.

[16] : Thomas Niemann. A GUIDE TO LEX & YACC. Portland, Oregon.

Références

[17] : Alexis Nasr Carlos Ramisch Sylvain Sené. Grammaires attribuées. Compilation – L3 Informatique Département Informatique et Interactions Aix Marseille Université.

[18] : Luigi Santocanale. Traduction et Sémantique Traduction dirigée par la syntaxe. LIF, Université de Provence Marseille, France. 19 mars 2010.

[19] : Claude Moulin. Théorie des Langages Analyse Sémantique. Université de Technologie de Compiègne Printemps 2013.

[20] : https://www.ulaval.ca/etudes/cours/ift-3101-compilation-et-interpretation

[21] : Fabienne Carrier. Structure d'un compilateur. Université Grenoble Alpes. 4 novembre 2016.

[22] : Rémi Forax. Génération de code. Université de Marne la Vallée.

[23] : Samuel Tardieu. Optimisation de code. École Nationale Supérieure des Télécommunications.

[24] : Christine Paulin-Mohring. Introduction à la compilation. Université Paris Sud Master Informatique 2011-2012.

[25] : Flavien Breuvart. Compilation Course. Institut Galilée - Université Paris 13. January 6, 2025.