République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Ecole Normale Supérieure Assia Djebar Constantine

Département de l'Informatique et des Mathématiques

3eme année Informatique

Support de cours

Module :

Compilation

Préparé par : Dr. BETTOU Farida
E-mail : bettou.farida@ensc.dz

Année 2023

Module : Compilation

Auteur : Dr. BETTOU Farida

Pré-requis : Théorie du langage

Description du module

Niveau : 3eme Année informatique

Volume horaire hebdomadaire

- Cours : 1h30
- Travaux Dirigés(TD) :1h30
- Travaux Pratique (TP) :1h30

Coefficient : 04

Objectifs pédagogiques

L'objectif d'un cours sur la compilation est de permettre aux étudiants de comprendre en profondeur le processus de compilation, qui est essentiel pour la création de logiciels. En somme, un cours de compilation vise à fournir aux étudiants les compétences et les connaissances nécessaires pour concevoir, créer et comprendre des compilateurs, ce qui est essentiel pour le développement de logiciels avancés et la compréhension approfondie du fonctionnement des langages de programmation.

Table des matières

CHAPITRE 0 2 : ANALYSE LEXICALE

CHAPITRE 03 : ANALYSE SYNTAXIQUE

CHAPITRE 04 : ANALYSE SYNTAXIQUE DESCENDANTE.

CHAPITRE 05 : ANALYSE SYNTAXIQUE ASCENDANTE.

CHAPITRE 06 : TRADUCTION DIRIGEE PAR LA SYNTAXE

CHAPITRE 07 : ANALYSE SEMANTIQUE

CHAPITRE 08 : ENVIRONNEMENTS D'EXECUTION

CHAPITRE 09 : PRODUCTION ET OPTIMISATION DE CODE

Liste des Figures

Liste des Tableaux

Chapitre 01 :
Introduction aux Compilateurs

1. Introduction

Les ordinateurs sont un mélange équilibré de logiciels et de matériel. Le matériel n'est qu'un élément mécanique et un logiciel compatible contrôle ses fonctions. Le matériel comprend les instructions sous forme de charge électronique, qui est la contrepartie du langage binaire dans la programmation logicielle. Le langage binaire n'a que deux alphabets, 0 et 1. Pour instruire, les codes matériels doivent être écrits au format binaire, qui est simplement une série de 1 et de 0. Ce serait une tâche difficile et fastidieuse pour les programmeurs informatiques d'écrire de tels codes, c'est pourquoi nous disposons de compilateurs pour écrire de tels codes. Nous avons appris que tout système informatique est constitué de matériel et de logiciels. Le matériel comprend un langage que les humains ne peuvent pas comprendre. Par conséquent, nous écrivons des programmes dans un langage de haut niveau, plus facile à comprendre et à retenir. Ces programmes sont ensuite intégrés à une série d'outils et de composants du système d'exploitation pour obtenir le code souhaité pouvant être utilisé par la machine. Ceci est connu sous le nom de système de traitement du langage.

Un compilateur est un logiciel essentiel dans le domaine de la programmation informatique. Il s'agit d'un outil qui transforme le code source écrit par un développeur dans un langage de programmation donné en un code binaire exécutable par une machine. Cette transformation se déroule en plusieurs étapes, chacune étant gérée par le compilateur.

2. Traducteur

Un traducteur est un programme qui prend en entrée un programme écrit dans une langue et produit en sortie un programme dans une autre langue. Outre la traduction du programme, le traducteur remplit un autre rôle très important : la détection des erreurs. Toute violation des spécifications du langage de haut niveau serait détectée et signalée aux programmeurs. Les rôles importants du traducteur sont :

• Traduire l'entrée du programme en langage de haut niveau en un programme en langage machine équivalent.

• Fournir des messages de diagnostic chaque fois que le programmeur viole la spécification du langage de haut niveau.

3. Type de traducteurs

a. Compilateur

b. Interpréteur

c. Préprocesseur

3.1 Compilateur

Un compilateur est un programme qui a comme entrée un code source écrit en langage de haut niveau (langage évolué) est produit comme sortie un code cible en langage de bas niveau (langage d'assemblage ou langage machine). La traduction ne peut être effectué que si le code source est correct car, s'il y a des erreurs, le rôle du compilateur se limitera à produire en sortie des messages d'erreurs (voir figure 1.1).

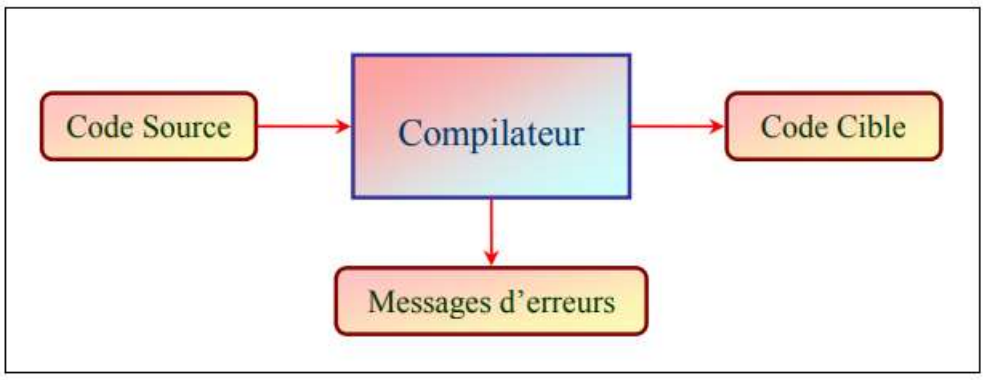


Figure 1.1 : Rôle du compilateur

Un compilateur est donc un traducteur de langage évolué qu'on ne doit pas confondre avec un interpréteur qui est un autre type de traducteur. En effet, au lieu de produire un programme cible comme dans le cas d'un compilateur, un interpréteur exécute lui-même au fur et à mesure les opérations spécifiées par le programme source. Il analyse une instruction après l'autre puis

l'exécute immédiatement. A l'inverse d'un compilateur, il travaille simultanément sur le programme et sur les données. L'interpréteur doit être présent sur le système à chaque fois que le programme est exécuté, ce qui n'est pas le cas avec un compilateur. Généralement, les interpréteurs sont assez petits, mais le programme est plus lent qu'avec un langage compilé. Un autre inconvénient des interpréteurs est qu'on ne peut pas cacher le code, et donc garder des secrets de fabrication : toute personne ayant accès au programme peut le consulter et le modifier comme elle le veut. Par contre, les langages interprétés sont souvent plus simples à utiliser et tolèrent plus d'erreurs de codage que les langages compilés. Des exemples de langages interprétés sont : BASIC, scheme, CaML, Tcl, LISP, Perl, Prolog Il existe des langages qui sont à mi-chemin de l'interprétation et de la compilation. On les appelle langages Pcode ou langages intermédiaires. Le code source est traduit (compilé) dans une forme binaire compacte (du pseudo-code ou p-code) qui n'est pas encore du code machine. Lorsqu'on exécute le programme, ce P-code est interprété. Par exemple en Java, le programme source est compilé pour obtenir un fichier (.class) « byte code » qui sera interprété par une machine virtuelle. Un autre langage p-code : Python.

3.2 Interpréteur

Un interpréteur est un logiciel dont la fonction est très similaire à celle d'un compilateur. L'entrée d'un interprète est un programme écrit dans un langage de haut niveau, mais plutôt que de générer un programme en langage machine, l'interpréteur effectue en réalité les calculs spécifiés dans le programme source. En d'autres termes, la sortie d'un compilateur est un programme, tandis que la sortie d'un interpréteur est la sortie du programme source. La figure 1.2 montre que même si les entrées peuvent être identiques, les compilateurs et les interprètes produisent des sorties très différentes. Néanmoins, bon nombre des techniques utilisées dans la conception des compilateurs sont également applicables aux interprètes.

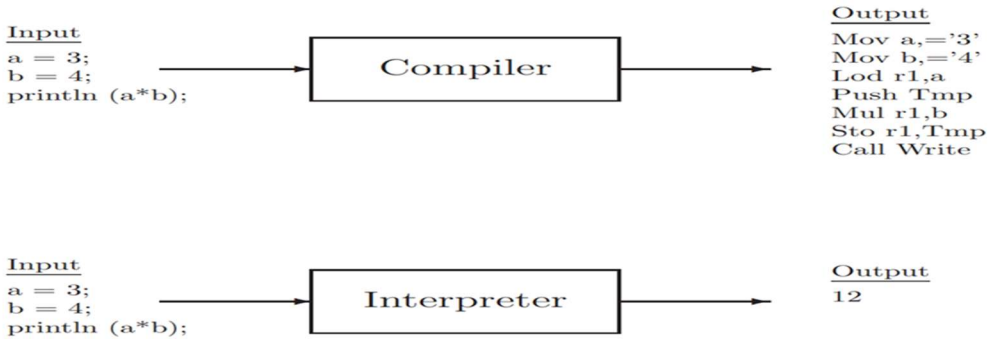


Figure 1.2 : Un compilateur et un interprète produisent des sorties très différentes pour la même entrée

3.3 Préprocesseur

Un préprocesseur produit des entrées pour les compilateurs. Ils peuvent remplir les fonctions suivantes. • Traitement des macros : un préprocesseur peut permettre à un utilisateur de définir des macros courtes pour des constructions plus longues. • Inclusion de fichiers : un préprocesseur peut inclure des fichiers d'en-tête dans le texte du programme. • Préprocesseur rationnel : ces préprocesseurs complètent les langages plus anciens avec des fonctionnalités de contrôle de flux et de structuration de données plus modernes. • Extensions de langage : ces préprocesseurs tentent d'ajouter des fonctionnalités au langage dans une certaine mesure à la macro intégrée.

4. Structure générale d'un compilateur

Un compilateur est un logiciel essentiel dans le domaine de la programmation informatique. Il s'agit d'un outil qui transforme le code source écrit par un développeur dans un langage de programmation donné en un code binaire exécutable par une machine. Cette transformation se déroule en plusieurs étapes, chacune étant gérée par le compilateur. Voici une brève explication des principales étapes du processus de compilation:

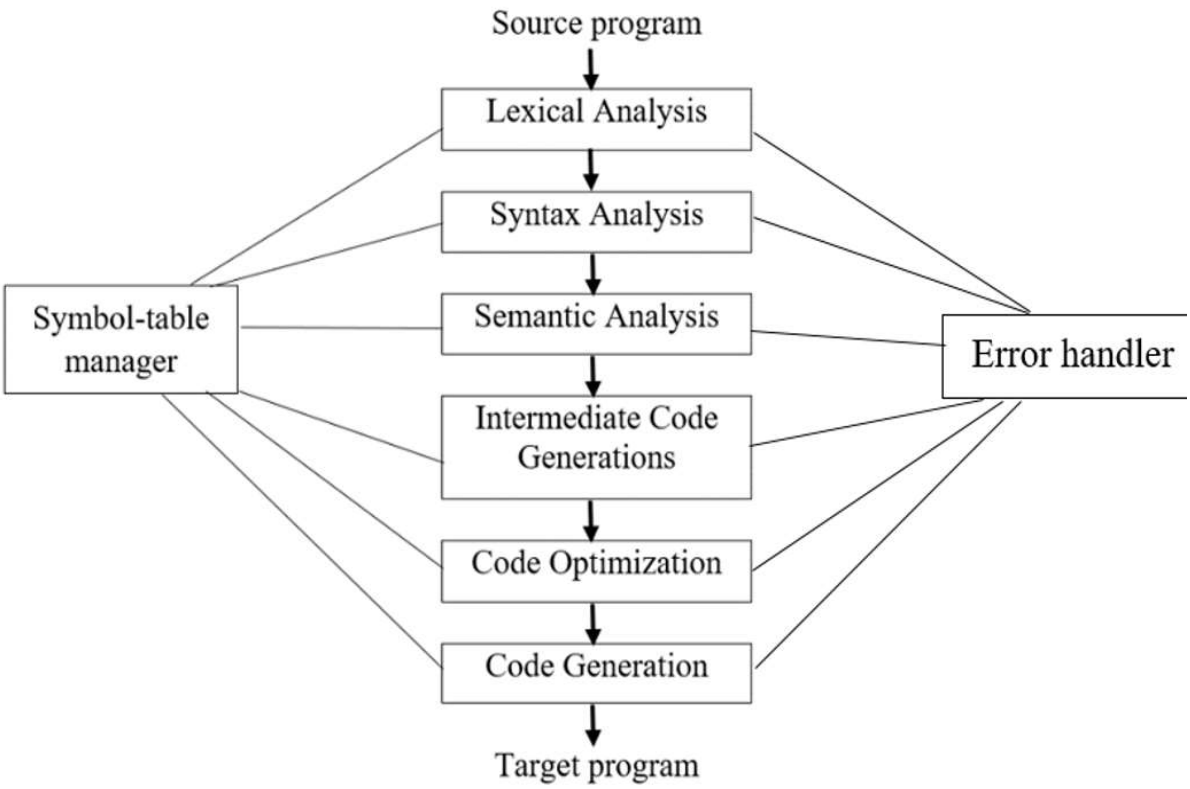


Figure 1.3 : Phases et modules de compilation

4.1 Analyse lexicale : La première étape consiste à analyser le code source pour identifier et regrouper les éléments lexicaux, tels que les mots-clés, les opérateurs, les constantes et les identificateurs.

4.2 Analyse syntaxique : Une fois les éléments lexicaux identifiés, l'analyse syntaxique vérifie que le code source suit les règles de syntaxe du langage de programmation. Elle crée également une structure de données appelée "arbre syntaxique" pour représenter la structure hiérarchique du code.

4.3 Analyse sémantique : Cette étape vérifie que le code source est cohérent sur le plan sémantique. Elle s'assure que les opérations sont effectuées sur des types de données appropriés et que toutes les variables sont déclarées avant leur utilisation.

4.4 Génération de code intermédiaire : Après avoir validé la sémantique, le compilateur génère un code intermédiaire. Ce code est une représentation abstraite du programme source qui peut être plus facilement optimisée et traduite en code machine.

4.5 Optimisation du code : Le compilateur peut appliquer diverses optimisations sur le code intermédiaire pour améliorer l'efficacité de l'exécution du programme final. Cela comprend l'élimination de code mort, la réduction des opérations redondantes, etc.

4.6 Génération de code machine : Enfin, le compilateur génère le code binaire exécutable à partir du code intermédiaire optimisé. Ce code est spécifique à l'architecture matérielle de la machine cible.

Une fois que le compilateur a terminé ces étapes, le programme est prêt à être exécuté sur l'ordinateur cible. Les compilateurs sont essentiels pour la programmation car ils permettent aux développeurs de travailler dans des langages de haut niveau, plus abstraits et plus faciles à lire et à comprendre, tout en garantissant que ces programmes puissent être exécutés efficacement par des ordinateurs.

5. Différence entre compilateur et interpréteur

Les compilateurs et les interprètes sont deux approches distinctes pour exécuter des programmes écrits dans des langages de programmation. Voici les principales différences entre les deux :

- **Processus de traduction** :
  - *Compilateur* : Un compilateur traduit l'ensemble du code source en langage machine ou en code intermédiaire en une seule fois, généralement avant l'exécution. Le résultat est généralement un fichier exécutable indépendant.
  - *Interpréteur* : Un interpréteur lit et traduit le code source ligne par ligne ou bloc par bloc au fur et à mesure de l'exécution. Il n'y a pas de fichier exécutable distinct généré.
- **Vitesse d'exécution** :
  - *Compilateur* : Les programmes compilés ont tendance à s'exécuter plus rapidement que les programmes interprétés, car la traduction est effectuée une seule fois avant l'exécution.
  - *Interpréteur* : Les programmes interprétés sont généralement plus lents car la traduction a lieu en temps réel pendant l'exécution.
- **Portabilité** :
  - *Compilateur* : Les programmes compilés sont souvent spécifiques à une architecture matérielle ou à une plate-forme. Ils doivent être recompilés pour chaque architecture cible.
  - *Interpréteur* : Les programmes interprétés sont généralement plus portables, car l'interpréteur s'adapte à l'architecture matérielle sur laquelle il s'exécute, ce qui réduit le besoin de recompilation.
- **Débogage** :
  - *Compilateur* : Le débogage peut être plus difficile avec les programmes compilés car les erreurs ne sont détectées qu'après la compilation. Vous devez souvent examiner le code machine généré pour identifier les problèmes.
  - *Interpréteur* : Le débogage est souvent plus convivial avec les interpréteurs car les erreurs sont généralement signalées au fur et à mesure de leur apparition, et vous avez accès au code source d'origine.

- **Exemples de langages** :
  - *Compilateur* : Des langages comme C, C++, Rust sont généralement compilés.
  - *Interpréteur* : Des langages comme Python, Ruby, JavaScript sont généralement interprétés.

Ces différences influencent le choix entre l'utilisation d'un compilateur ou d'un interpréteur en fonction des besoins d'un projet particulier. Les compilateurs sont souvent utilisés lorsque les performances sont cruciales, tandis que les interpréteurs sont préférés pour la facilité de développement et la portabilité.

Chapitre 02 :
Analyse Lexicale

1. Aperçu de l'analyse lexicale

L'analyse lexicale est la première étape du processus de compilation d'un programme informatique. Son rôle principal est de lire le code source du programme et de le diviser en unités lexicales ou tokens. Ces tokens sont des éléments individuels du code, tels que des mots-clés, des identificateurs, des opérateurs, des constantes, et d'autres symboles.

2. Rôle de l'analyseur lexical

Il s'agit de transformer des suites de caractères du code source en suite de symboles, correspondant aux unités lexicales, que l'analyseur lexical produit comme résultat de l'analyse. Pour cela, il existe deux approches possibles sachant qu'une passe est définie comme correspondant à un traitement entier d'une représentation du programme source pour produire une représentation équivalente en mémoire secondaire :

• L'analyseur lexical effectue un traitement de la totalité du code source en une première passe, ce qui lui permet d'en obtenir une représentation équivalente sous forme d'une suite d'unités lexicales, sauvegardée dans un fichier en mémoire secondaire. Ce fichier sera l'entrée de l'analyseur syntaxique, qui accomplira son travail pendant une deuxième passe.

• L'analyseur lexical fonctionne comme une procédure sollicitée à chaque fois, par l'analyseur syntaxique, pour lui fournir une unité lexicale. L'analyseur lexical effectue, pour cela, son travail, pas à pas, en synchronisation avec l'avancement de l'analyse syntaxique. On dit que l'analyse lexicale et syntaxique partagent une même passe. Dans ce cas, il n'est plus question de sauvegarder les unités dans un fichier intermédiaire, cependant, les deux analyseurs (lexical et syntaxique) doivent se trouver ensemble en mémoire.

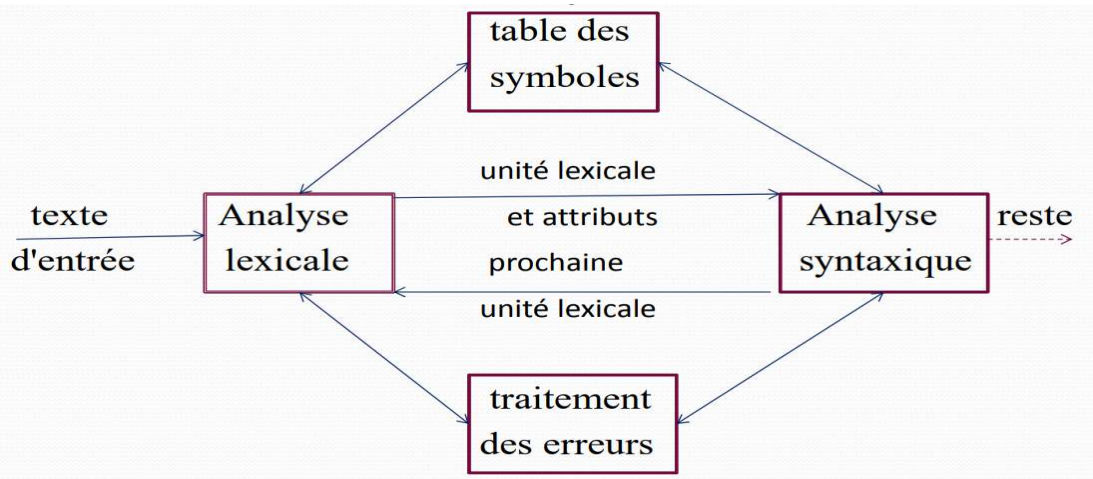


Figure 2.1 : Rôle de l'analyseur lexical

3. Tâches effectuées par l'analyseur lexical

L'analyse lexicale effectue un ensemble de tâches :

1- Lecture du fichier du code source (fichier texte) caractère par caractère avec éventuellement l'élimination de caractères et informations inutiles (blancs, tabulations, commentaires, caractère de fin de ligne, …) pour réduire la représentation du programme. Une erreur est signalée à ce niveau si un caractère non permis (illégal) par le langage est rencontré.

2- Concaténation des caractères pour former des unités lexicales et signaler, éventuellement, une erreur si la chaîne dépasse une certaine taille.

3- Association d'un symbole à chaque unité lexicale. Cette opération peut engendrer une erreur si l'unité formée ne correspond à aucune unité légale du langage.

4- Enregistrement de chaque unité lexicale et des informations la concernant (nom, valeur, …) dans la table des symboles (en invoquant le gestionnaire de la table des symboles) ou dans un fichier résultat. A ce niveau, il est possible de détecter certaines erreurs telles que la double déclaration d'un identificateur, l'utilisation d'un identificateur sans déclaration préalable, etc.

5- Création d'une liaison entre les unités lexicales d'une ligne et la ligne du fichier la contenant. Cette opération, qui se fait par une simple numérotation des lignes du texte du code

source, permet la localisation des lignes en cas d'erreur. Dans certains compilateurs, l'analyseur lexical est chargé de créer une copie du programme source en y intégrant les messages d'erreurs lexicales. Les symboles associés aux unités lexicales correspondent à la nature de celles-ci. On distingue plusieurs catégories de symboles :

1- Les mots réservés du langage (if, then, begin pour le langage Pascal, par exemple)
2- Les constantes (3, 3.14, True, 'Bonjour', …)
3- Les identificateurs qui peuvent être des noms de variables, de fonctions, de procédures, etc.
4- Les symboles spéciaux, en tenant compte des possibilités d'assemblage de caractères tels que < > <= <> : := + - /*
5- Les séparateurs tels que ; (point virgule) et . (point).

4. Spécification des unités lexicales

L'ensemble des unités lexicales, qu'un analyseur reconnaît, constitue un langage régulier L qui peut être décrit par des expressions régulières et reconnu par un automate d'états fini. La spécification ou description des unités lexicales peut donc être effectuée en utilisant une notation appelée expressions régulières. Une expression régulière est construite à partir d'expressions régulières plus simples, en utilisant un ensemble de règles de définition qui spécifient comment le langage est formé. Soit le vocabulaire $\sum$ = {a, b, c}

1- ε dénote la chaîne vide
2- L'expression régulière a/b dénote l'ensemble des chaînes {a,b}
3- L'expression a* dénote l'ensemble de toutes les chaînes formées d'un nombre quelconque de a (pouvant être nul), c à d {ε, a, aa, aaa, …}
4- L'expression a+ dénote l'ensemble de toutes les chaînes formées d'un nombre quelconque, non nul, de a, c à d {a, aa, aaa, …}. Si r est une expression régulière alors r* = r+/ε et r+ = rr*
5- La notation r? est une abréviation de r/ε et signifie zéro ou une instance de r
6- La notation [abc], où a, b, c sont des symboles de l'alphabet, dénote l'expression régulière a/b/c. Une classe de caractères [a-z] signifie a/b/c/d/ …/z

Exemple 1

Pour chacune des expressions régulières ri suivantes, on souhaite déterminer le langage dénoté par ri .

r1 = (a/b)(a/b)

r2 = (a/b)*

r3 = (a*b*)*

r4 = a/a*b

Chaque langage dénoté par ri est noté L(ri)

L(r1) = {aa, ab, ba, bb}

L(r2) = {ε, a, aa, aaa, …., b, bb, bbb, …, ab, aab, …, abb, abbb, ba, bba, bbaa, aba, …}.

C'est l'ensemble de toutes les chaînes composées d'un nombre quelconque de a et d'un nombre quelconque de b.

L(r3) = L(r2)

L(r4) = {a, b, ab, aab, aaab, …, aaaa…ab}

En plus de la chaîne a, cet ensemble contient des chaînes composées d'un nombre quelconque non nul de a suivis par un b.

Exemple 2

On souhaite déterminer la définition régulière de l'ensemble des identificateurs d'un langage sachant qu'ils sont constitués de suite de lettres et de chiffres commençant par une lettre. Cette définition régulière peut être donnée par :

• lettre = A/B/…/Z/a/b/…/z ou bien [A-Za-z]

• chiffre = 0/1/2/3/4/5/6/7/8/9 ou bien [0-9]

• ident = lettre (lettre/chiffre)* qu'on peut noter directement ident = [A-Za-z] [A-Za-z0-9]*

Remarque

Les expressions régulières sont un outil puissant et pratique pour définir les constituants élémentaires des langages de programmation. Cependant, le pouvoir descriptif des expressions régulières est limité car elles ne peuvent pas être utilisées dans certains cas tels que :

- La définition des constructions équilibrées ou imbriquées. Par exemple, pour les chaînes contenant obligatoirement pour chaque parenthèse ouvrante une parenthèse fermantes, les expressions régulières ne permettent pas d'assurer cela, on dit qu'elles ne savent pas compter.
- La définition d'expressions contenant des répétitions de sous chaînes à différente positions (par exemple des chaînes de la forme αcαdeα où α est une combinaison des caractères a et b).
- La définition d'expressions contenant à différentes positions des séquences de caractères ayant la même longueur (par exemple an bn où n désigne le nombre de répétitions du caractère qui le précède). On peut, cependant, utiliser les expressions régulières dans le cas d'un nombre fixe de répétitions d'une construction donnée.

5. Expressions régulières

L'expression régulière est une formule qui décrit un ensemble possible de chaînes. Composant de l'expression régulière.

X	le caractere x
.	n'importe quel caractère, accepte généralement une nouvelle ligne
[x y z]	l'un des caractères x, y, z, …..
R?	un R ou rien (=éventuellement comme R)
R*	zéro ou plusieurs occurrences…..
R+	une ou plusieurs occurrences ……
R1R2	un R1 suivi d'un R2
R2R1	soit un R1, soit un R2.

Un jeton est soit une chaîne unique, soit l'un d'un ensemble de chaînes d'un certain type. Si nous considérons l'ensemble de chaînes dans chaque classe de jetons comme un langage, nous pouvons utiliser la notation d'expression régulière pour décrire les jetons. Considérons un identifiant, qui est défini comme une lettre suivie de zéro ou plusieurs lettres ou chiffres. En notation d'expression régulière, nous écririons.

Identifiant = lettre (lettre | chiffre)*

Voici les règles qui définissent l'expression régulière sur l'alphabet.

• est une expression régulière désignant { € }, le langage contenant uniquement le chaîne vide.

• Pour chaque « a » dans ∑, est une expression régulière désignant { a }, le langage avec seulement une chaîne composée du seul symbole « a ». • Si R et S sont des expressions régulières, alors

(R) | (S) signifie LrUL

R.S signifie Lr. Ls

R* désigne Lr*

5.1 Operations

Les différentes opérations sur les langues sont :

- L'union de deux langues L et M s'écrit L U M = {s | s est dans L ou s est dans M}
- La concaténation de deux langages L et M s'écrit LM = {st | s est dans L et t est dans M}
- La fermeture Kleene d'un langage L s'écrit L* = Zéro ou plusieurs occurrences du langage L.

5.2 Notations

Si r et s sont des expressions régulières désignant les langages L(r) et L(s), alors

• Union : (r)|(s) est une expression régulière désignant L(r) U L(s)

• Concaténation : (r)(s) est une expression régulière désignant L(r)L(s)

• Fermeture de Kleene : (r)* est une expression régulière désignant (L(r))*

• (r) est une expression régulière désignant L(r)

5.3 Préséance et associativité

- *, concaténation (.) et | (signe de tuyau) restent associatifs
- * a la priorité la plus élevée
- La concaténation (.) a la deuxième priorité la plus élevée.
- | (signe de tuyau) a la priorité la plus basse de toutes.

5.4 Représenter les jetons valides d'un langage dans une expression régulière

Pour représenter l'occurrence de symboles à l'aide d'expressions régulières, vous pouvez utiliser des quantificateurs spéciaux. Les quantificateurs vous permettent de spécifier combien de fois un symbole ou un groupe de symboles doit apparaître dans une chaîne de caractères. Voici quelques exemples de quantificateurs couramment utilisés avec des symboles :

a. **Un symbole unique** :
   - Pour représenter un symbole unique, vous n'avez besoin d'aucun quantificateur. Par exemple, l'expression régulière a correspondra à la lettre "a" dans un texte.

b. **Zéro ou une occurrence** :
   - Le quantificateur "?" signifie "zéro ou une occurrence" du symbole précédent. Par exemple, l'expression régulière a? correspondra à "a" s'il est présent ou à une chaîne vide si "a" est absent.

c. **Zéro ou plusieurs occurrences** :
   - L'astérisque "*" signifie "zéro ou plusieurs occurrences" du symbole précédent. Par exemple, l'expression régulière ab*c correspondra à "ac," "abc," "abbc," etc.

d. **Une ou plusieurs occurrences** :
   - Le signe plus "+" signifie "une ou plusieurs occurrences" du symbole précédent. Par exemple, l'expression régulière ab+c correspondra à "abc," "abbc," etc., mais pas à "ac."

e. **Un nombre précis d'occurrences** :

- Les accolades "{}" vous permettent de spécifier un nombre précis d'occurrences. Par exemple, l'expression régulière `a{3}` correspondra uniquement à "aaa."

f. **Un nombre minimum et maximum d'occurrences** :

- Les accolades avec deux nombres séparés par une virgule permettent de spécifier un nombre minimum et maximum d'occurrences. Par exemple, l'expression régulière `a{2,4}` correspondra à "aa," "aaa," ou "aaaa," mais pas à "a" ou "aaaaa."

g. **Un nombre exact ou un intervalle précis d'occurrences** :

- Vous pouvez également spécifier un nombre exact d'occurrences en utilisant simplement la notation avec des accolades. Par exemple, `a{3}` correspondra à "aaa."

• [a-z] est constitué de tous les alphabets minuscules de la langue anglaise.
• [A-Z] correspond à tous les alphabets majuscules de la langue anglaise.
• [0-9] sont des chiffres entièrement naturels utilisés en mathématiques.

Ces quantificateurs peuvent être utilisés avec des symboles simples ou avec des groupes de symboles dans des expressions régulières plus complexes. Ils permettent de décrire avec précision le modèle d'occurrence des symboles que vous recherchez dans un texte. N'hésitez pas à les combiner avec d'autres éléments d'expression régulière pour créer des motifs plus complexes selon vos besoins.

5. 5 Représenter l'occurrence de symboles à l'aide d'expressions régulières

letter = A/B/…/Z/a/b/…/z or [A-Za-z]
digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 or [0-9]
sign = [+ | -]

5. 6 Représentation des jetons de langage à l'aide d'expressions régulières

Décimal = (signe) ? (chiffre)+
ident = lettre (lettre/chiffre)* ou ident = [A-Za-z] [A-Za-z0-9]*

Le seul problème qui reste avec l'analyseur lexical est de savoir comment vérifier la validité d'une expression régulière utilisée pour spécifier les modèles de mots-clés d'une langue. Une solution bien acceptée consiste à utiliser des automates finis pour la vérification.

6. Approches de construction d'analyseur lexical

La construction d'analyseurs lexicaux est une étape cruciale dans le processus de compilation d'un langage de programmation. Elle consiste à analyser le code source en entrée et à le découper en unités lexicales (ou tokens) telles que des mots-clés, des identificateurs, des

opérateurs, des constantes, etc. Il existe plusieurs approches pour construire des analyseurs lexicaux, notamment :

a) Utilisation d'expressions régulières :

L'approche la plus courante consiste à utiliser des expressions régulières pour définir les motifs de chaque type de token dans le langage. Chaque expression régulière décrit la structure d'un type de token, et un analyseur lexical utilise ces expressions pour identifier et extraire les tokens du code source.

b) Automates finis déterministes (AFD) :

Les AFD sont des machines à états finis qui peuvent être utilisées pour reconnaître des motifs lexicaux dans le code source. Chaque état de l'automate représente une position possible dans le motif en cours de recherche, et les transitions entre les états sont déclenchées par les caractères lus. Les AFD sont souvent générés automatiquement à partir d'expressions régulières.

c) Analyseurs lexicaux générés automatiquement :

Il existe des outils et des générateurs d'analyseurs lexicaux, tels que Lex et Flex, qui prennent en entrée une spécification des motifs lexicaux en utilisant des expressions régulières et génèrent automatiquement le code d'un analyseur lexical.

d) Analyseurs manuels :

Dans certains cas, les développeurs peuvent choisir de créer des analyseurs lexicaux manuellement en écrivant du code pour reconnaître les tokens un par un. Cela peut être nécessaire lorsque la structure du langage est complexe ou que les besoins sont très spécifiques.

e) Analyseurs récursifs descendants :

Dans certaines situations, les analyseurs lexicaux sont intégrés directement dans les analyseurs syntaxiques. Cette approche est couramment utilisée dans les analyseurs récursifs descendants, où les tokens sont générés à la volée pendant l'analyse syntaxique.

f) Analyseurs à deux passes :

Dans certains langages, il est nécessaire de réaliser une première passe pour identifier les tokens, suivie d'une deuxième passe pour l'analyse syntaxique. Cette approche est utilisée lorsque la syntaxe du langage est ambiguë ou complexe.

Le choix de l'approche dépend souvent de la complexité du langage de programmation à analyser, des exigences de performances et des outils disponibles. Les expressions régulières et les outils de génération automatique d'analyseurs lexicaux sont couramment utilisés car ils simplifient grandement le processus de construction d'analyseurs lexicaux et garantissent une implémentation précise des spécifications lexicales du langage.

6.1 Diagramme de transition

Un diagramme de transition comporte une collection de nœuds ou de cercles, appelés états. Chaque état représente une condition qui pourrait se produire pendant le processus d'analyse de l'entrée à la recherche d'un lexème correspondant à l'un des nombreux modèles. Les arêtes sont dirigées d'un état du diagramme de transition à un autre. Chaque arête est étiquetée par un symbole ou un ensemble de symboles. Si nous sommes dans un état s et que le symbole d'entrée suivant est a, nous recherchons une arête hors de l'état s étiquetée par a. Si nous trouvons une telle arête, nous avançons le pointeur vers l'avant et entrons dans l'état du diagramme de transition auquel cette arête mène.

Certaines conventions importantes concernant les diagrammes de transition sont :

1. Certains États sont dits acceptants ou définitifs. Ces états indiquent qu'un lexème a été trouvé, bien que le lexème réel puisse ne pas comprendre toutes les positions entre les pointeurs Begin et Forward du lexème, nous indiquons toujours un état d'acceptation par un double cercle.
2. De plus, s'il est nécessaire de ramener le pointeur avant à une position, nous placerons en plus un * à proximité de cet état d'acceptation.
3. Un état est conçu comme l'état ou l'état initial ; il est indiqué par un bord intitulé « début » entrant de nulle part. Le diagramme de transition commence toujours dans l'état précédant l'utilisation des symboles d'entrée.

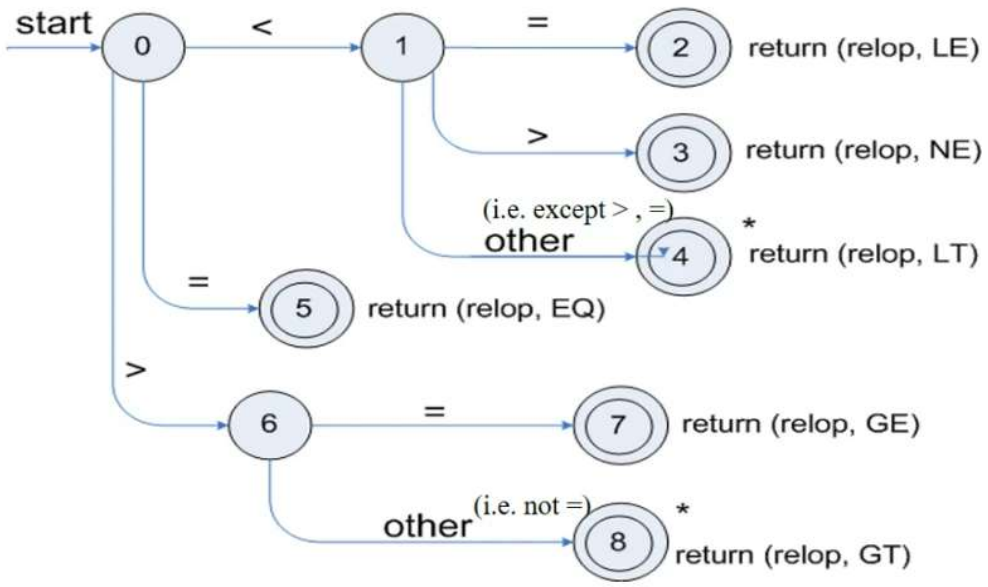


Figure 2.2 : Exemple de diagrammes de transition

Comme étape intermédiaire dans la construction d'un Analyseur Lexical, nous produisons d'abord un organigramme stylisé, appelé diagramme de transition. Les positions dans un diagramme de transition (DT) sont dessinées sous forme de cercles et sont appelées états.

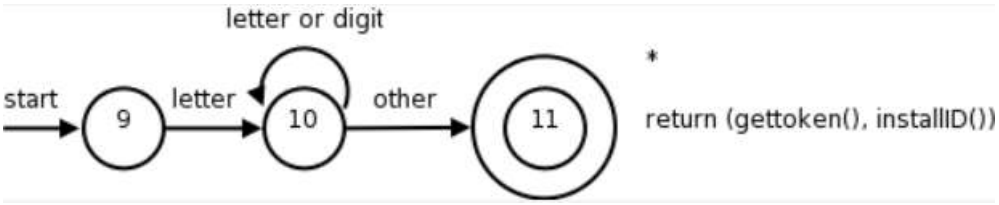


Figure 2.3 : Un diagramme de transition pour les identifiants et les mots-clés

Le diagramme de transition ci-dessus pour un identifiant est défini comme étant une lettre suivie d'un nombre quelconque de lettres ou de chiffres. Une séquence de diagrammes de transition peut être convertie dans le programme pour rechercher les jetons spécifiés par les diagrammes. Chaque état reçoit un segment de code.

Exemple

On souhaite donner la définition régulière ainsi que le diagramme de transition qui représente l'ensemble des nombres réels non signés comme 2006, 12.33, 314.3E-2, 0.314E+1, 0.314E2, 314E-2

La définition régulière est donnée par :

• chiffre = 0/1/2/3/4/5/6/7/8/9 ou bien [0-9]

• chiffres = chiffre chiffre* ou bien chiffre+

• fractionOpt = .chiffres/ε ou bien (.chiffres)?

• exposantOpt= (E(+/-/ε) chiffres) / ε ou bien (E(+/-)?chiffres)?

• NbRéel = chiffres fractionOpt exposantOpt

Pour faciliter la représentation par un diagramme de transition, la spécification précédente peut être réécrite directement sous la forme :

NbRéel = chiffre chiffre* (. chiffre chiffre*)? (E(+/-)?chiffre chiffre*)?

Cette forme peut être représentée directement de la figure 2.4.

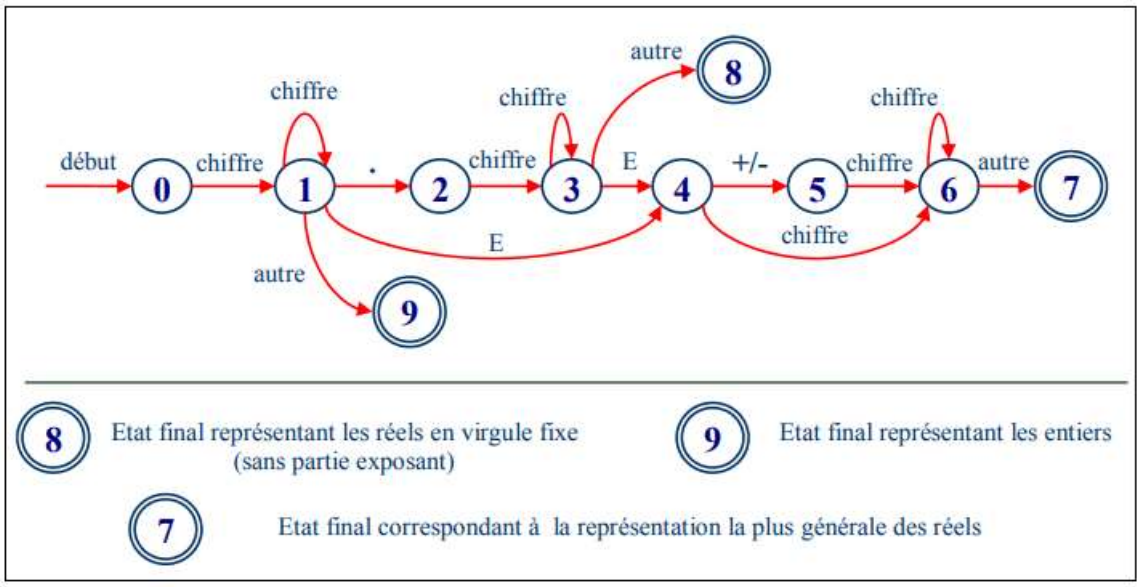


Figure 2.4. Diagramme de transition des nombres réels

6.2 Construction d'un analyseur lexical basé sur des automates finis

Les automates finis sont une machine à états qui prend une chaîne de symboles en entrée et change son état en conséquence. Les automates finis sont un outil de reconnaissance des expressions régulières. Lorsqu'une chaîne d'expression régulière est introduite dans des automates finis, elle change d'état pour chaque littéral. Si la chaîne d'entrée est traitée avec succès et que les automates atteignent leur état final, elle est acceptée, c'est-à-dire que la chaîne qui vient d'être alimentée est considérée comme un jeton valide du langage utilisé. Le modèle mathématique des automates finis se compose de :

- Ensemble fini d'états (Q)
- Ensemble fini de symboles d'entrée (Σ)
- Un état de démarrage (q0)
- Ensemble d'états finaux (qf)
- Fonction de transition (δ)

La fonction de transition (δ) mappe l'ensemble fini d'état (Q) à un ensemble fini de symboles d'entrée (Σ),

$$Q \times \Sigma \rightarrow Q$$

6.2.1 Démarche générale de construction d'un analyseur lexical

En se basant sur les propositions démontrables suivantes :

• L'ensemble des unités lexicales d'un langage donné constitue un langage régulier L.

• Pour toute expression régulière r, il existe un automate fini non déterministe (AFN) qui accepte l'ensemble régulier décrit par r.

• Si un langage L est accepté par un automate fini non déterministe (AFN), alors il existe automate fini déterministe (AFD) acceptant L. On peut définir une approche rigoureuse pour la construction d'un analyseur lexical en utilisant les automates d'états finis. Cette approche est constituée de 6 étapes :

Etape 1 : Spécification des unités lexicales Spécifier chaque type d'unité lexicale à l'aide d'une expression régulière (ER).

Etape 2 : Conversion ER en AFN Convertir chaque expression régulière en un automate d'états fini (non déterministe).

Etape 3 : Réunion des AFNs Construire l'automate Union de tous les automates de l'étape 2 (on peut ajouter un nouvel état initial d'où partent un ensemble d'arcs étiquetés ε).

Etape 4 : Déterminisation ou transformation de l'AFN obtenu en AFD Rendre l'automate de l'étape 3 déterministe.

Etape 5 : Minimisation de l'AFD. Minimiser l'automate obtenu à l'étape 4.

6.2.2 Automate fini non déterministe

Un AFN est défini par :

• Un ensemble d'état E

• Un ensemble de symboles d'entrée ou alphabet Σ

• Un état initial e0

• Un ensemble F d'états finaux ou d'acceptation

• Une fonction de transition Transiter qui fait correspondre à chaque couple (état,symbole), un ensemble d'états.

Exemple Considérons l'AFN qui reconnaît le langage décrit par l'expression régulière (a|b)*abb (voir figure 2.5). Pour cet automate :

$E = \{0, 1, 2, 3\}$ $e0 = 0$ $\Sigma = \{a, b\}$ $F = \{3\}$

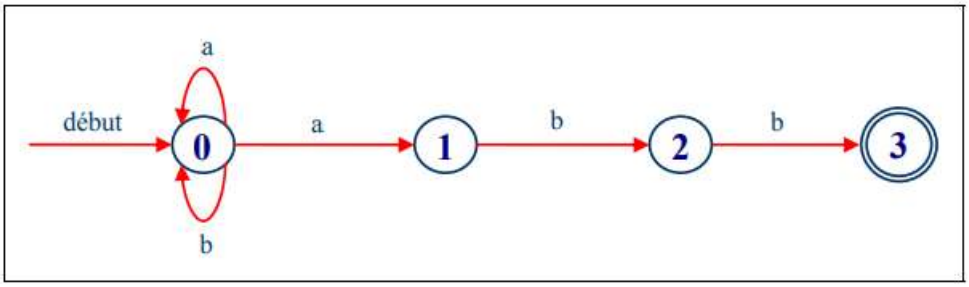


Figure 2.5. Un AFN pour l'expression (a|b)*abb

6.2.3 Automate fini déterministe

Un AFD est un cas particulier d'AFN dans lequel :

• Aucun état n'a de ε-transition

• Pour chaque état e et chaque symbole d'entrée a il y a au plus un arc étiqueté a qui quitte e Dans la table de transition d'un AFD, une entrée contient un état unique au maximum (les symboles d'entrée sont les caractères du texte source), il est donc très facile de déterminer si une chaîne est acceptée par l'automate vu qu'il n'existe, au plus, qu'un seul chemin entre l'état initial et un état final étiqueté par la chaîne en question.

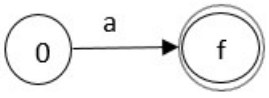
Remarque : La table de transition d'un AFN pour un modèle d'expression régulière peut être considérablement plus petite que celle d'un AFD. Cependant, l'AFD présente l'avantage de pouvoir reconnaître des modèles d'expressions régulières plus rapidement que l'AFN équivalent.

6.2.4 Construction d'AFN à partir d'expressions régulière

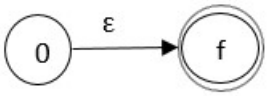
La construction d'un Automate à États Finis Non Déterministe (AFN) à partir d'une expression régulière est un processus qui permet de représenter formellement un langage régulier. Voici comment vous pouvez construire un AFN à partir d'une expression régulière :

a) Composants de base :

- Caractères uniques : chaque caractère de l'expression régulière correspond à un état dans le NFA. Par exemple, si votre expression régulière est « a », vous aurez un état intitulé « a ».

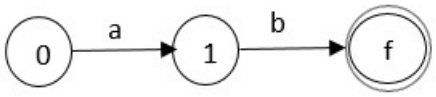


- Transitions Epsilon (transitions ε) : ce sont des transitions qui ne consomment aucune entrée. Ils sont utilisés pour gérer la concaténation. Représentez-les avec des flèches étiquetées ε.



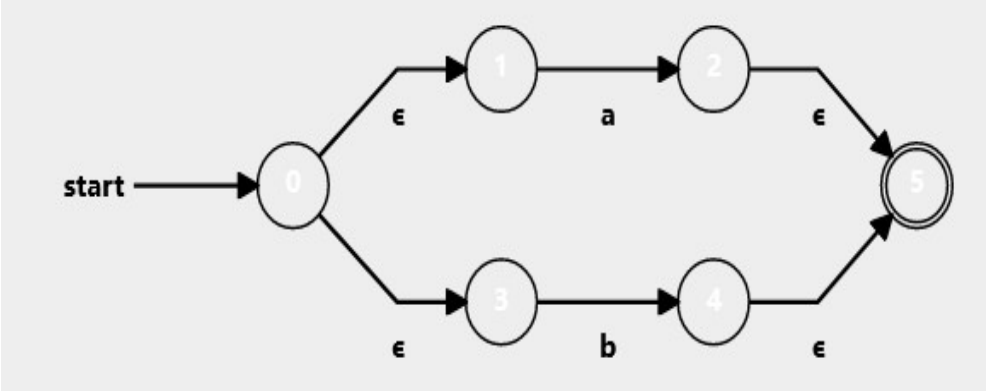
b) Concaténation :

- Si l'expression régulière est une concaténation d'expressions (par exemple, "ab"), créez une transition ε de l'état d'acceptation de la première expression à l'état initial de la deuxième expression.



c) Alternance (union) :

- Si l'expression régulière est une alternance (par exemple, "a|b"), créez un nouvel état initial et des ε-transitions de cet état initial aux états initiaux des expressions alternées.



d) Kleene Star (Clôture) :

- Si l'expression régulière a l'étoile de Kleene (par exemple, "a*"), créez un nouvel état initial, un nouvel état d'acceptation et des ε-transitions du nouvel état initial à l'état initial de l'expression répétée et du état d'acceptation de l'expression à la fois vers le nouvel état d'acceptation et vers l'état initial de l'expression.

e) Créer des États acceptants :

- S'il n'y a qu'une seule expression dans l'expression régulière, marquez son état final comme état d'acceptation.

f) Finalisation :

- S'il existe plusieurs expressions ou opérations, vous devriez maintenant avoir un NFA modifié. Vous pouvez le simplifier davantage en supprimant toutes les transitions ε (en effectuant des opérations de fermeture ε) si le langage le permet.

Ce processus construit un NFA qui reconnaît le langage défini par l'expression régulière donnée. La clé est de comprendre les opérations dans l'expression régulière (concaténation, alternance et fermeture) et de construire systématiquement les composants correspondants dans le NFA.

Exemple

L'AFN qui reconnaît l'expression régulière (a|b)*abb selon la méthode de construction de Thompson est donné dans la figure 2.6 :

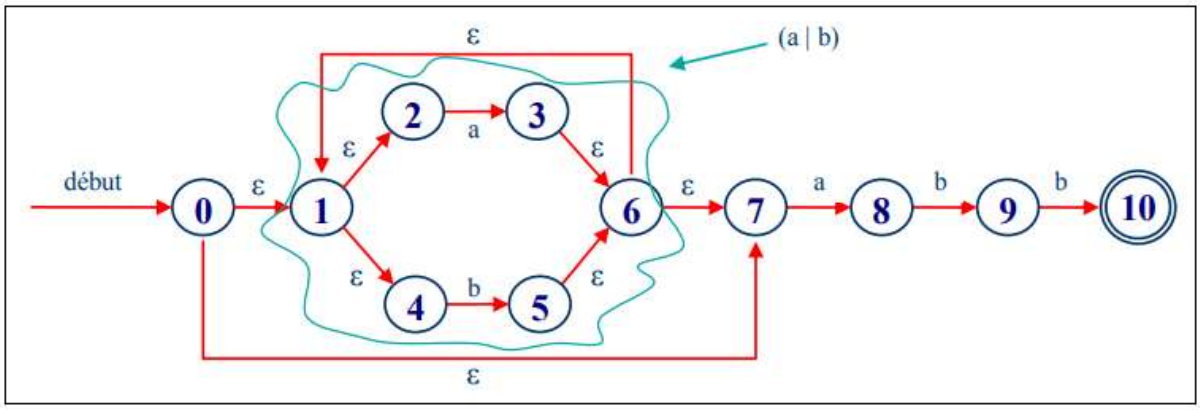


Figure 2.6. Automate fini non déterministe N pour accepter (a|b)*abb

6.2.5 Transition de l'AFN à l'AFD

La construction d'un Automate à États Finis Déterministe (AFD) à partir d'un Automate à États Finis Non Déterministe (AFN) est un processus appelé "détermination". Le but de la détermination est de transformer un AFN en un AFD équivalent qui reconnaît le même langage régulier. Voici un algorithme général pour construire un AFD à partir d'un AFN :

Entrée : Un AFN $(Q, \Sigma, \delta, q_0, F)$.

Sortie : Un AFD équivalent $(Q', \Sigma, \delta', q_0', F')$.

a) **Créer l'ensemble d'états initial du nouvel AFD** :

L'ensemble d'états initial du nouvel AFD sera l'ε-fermeture de l'état initial de l'AFN. Calculez l'ensemble d'états atteignables à partir de q_0 en suivant les transitions ε de l'AFN. Appelez cet ensemble Q'_0.

b) **Initialisation** :

Initialisez un ensemble de nouveaux états Q' avec Q'_0 comme unique élément.

Initialisez un ensemble d'états finaux F' vide.

c) **Boucle principale** :

Tant qu'il existe un nouvel ensemble d'états non traité dans Q', faites ce qui suit : a. Sélectionnez un ensemble d'états non traité Q'_x de Q'. b. Pour chaque symbole σ de l'alphabet Σ, calculez l'ensemble d'états accessible depuis Q'_x en suivant les transitions σ de l'AFN. Appelez cet ensemble $Q'_x\sigma$. c. Calculez l'ε-fermeture de $Q'_x\sigma$ en incluant tous les états accessibles à partir de $Q'_x\sigma$ en suivant les transitions ε de l'AFN. Appelez cet ensemble $Q'_x\sigma\varepsilon$. d. Si $Q'_x\sigma\varepsilon$ n'est pas déjà dans Q', ajoutez-le à Q' et répétez la boucle. e. Créez une transition de Q'_x vers $Q'_x\sigma\varepsilon$ en utilisant σ dans δ'. f. Si $Q'_x\sigma\varepsilon$ contient au moins un état final de l'AFN, ajoutez Q'_x à F'.

d) **Construction de l'AFD résultant** :

L'AFD résultant est composé de l'ensemble d'états Q', l'alphabet Σ, les transitions δ', l'état initial q_0' qui est Q'₀, et l'ensemble d'états finaux F'.

e) **Résultat** :

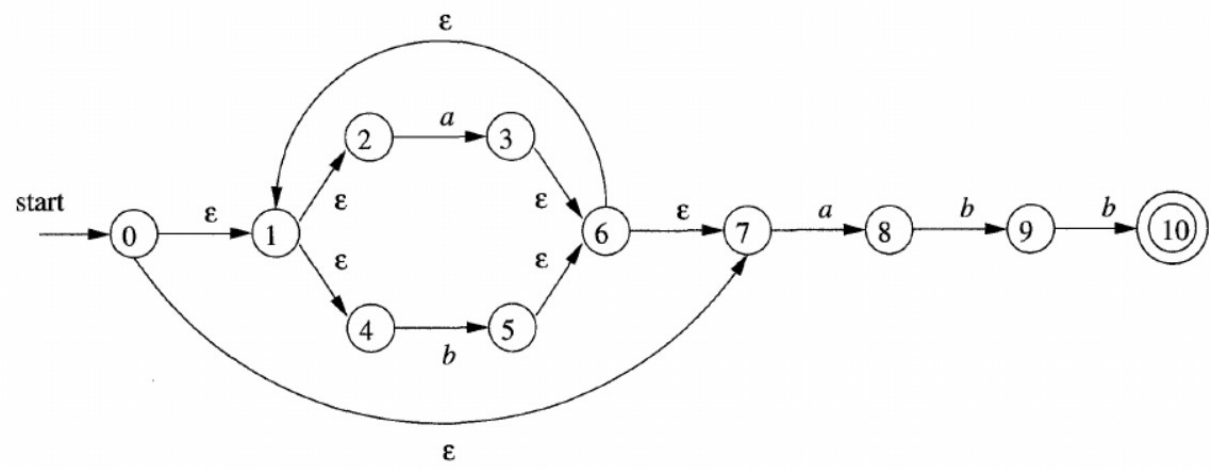
Le nouvel AFD ainsi construit est équivalent à l'AFN d'origine.

L'algorithme de détermination est basé sur la notion d'ensemble d'états atteignables dans l'AFN et l'utilisation de l'ε-fermeture pour gérer les transitions ε. Il garantit que l'AFD résultant reconnaîtra le même langage que l'AFN initial.

en utilisant l'algorithme de construction de sous-ensembles qui se présente comme suit :

```
Algorithme ConstructionSousEnsembles ;
Début
  ε-fermeture(e0) est l'unique état de Détats et il est non marqué;
  Tantque il existe un état non marqué dans Détats Faire
  Début
     marquer T ;
     Pour chaque symbole d'entrée a faire
     Début
          U ← ε-fermeture(Transiter(T,a)) ;
          Si U∉ Détats Alors Ajouter U comme nœud non marqué à Détats
          DTrans[T, a] ← U ;
     Fin ;
  Fin ;
Fin ;
```

Exemple d'application : soit l'AFN suivant qui accepte le langage (a|b)*abb



L'état de départ de l'AFD A=ε-f(0)={0,1,2,4,7}

- L'alphabet Σ = {a, b}
- B=ε-f(tran(A,a))=ε-f({3,8})={3,1,2,4,6,7,8} => Dtrans(A,a)=B.
- C=ε-f(tran(A,b))=ε-f({5})={1,2,4,5,6,7} => Dtrans(A,b)=C.
- ε-f(tran(B,a))=ε-f({3,8})={3,1,2,4,6,7,8} => Dtrans(B,a)=B.
- ε-f(tran(B,b))=ε-f({5,9})={1,2,4,5,6,7,9} => Dtrans(B,b)=D.
- ε-f(tran(C,a))=ε-f({3,8})={3,1,2,4,6,7,8} => Dtrans(C,a)=B.
- ε-f(tran(C,b))=ε-f({5})={1,2,4,5,6,7} => Dtrans(C,b)=C.
- ε-f(tran(D,a))=ε-f({3,8})={3,1,2,4,6,7,8} => Dtrans(D,a)=B.
- ε-f(tran(D,b))=ε-f({5,10})={1,2,4,5,6,7,10} => Dtrans(D,b)=E.
- ε-f(tran(E,a))=ε-f({3,8})={3,1,2,4,6,7,8} => Dtrans(E,a)=B.
- ε-f(tran(E,b))=ε-f({5})={1,2,4,5,6,7} => Dtrans(E,b)=C.

L'état E est l'état de fin de l'AFD car E contient l'état 10 qui est l'état d'acceptation de l'AFN.

Règle : un état de l'AFD est un état d'acceptation si c'est un ensemble d'états de l'AFN qui contient au moins un état d'acceptation de l'AFN.

L'AFD équivalent à l'AFN est :

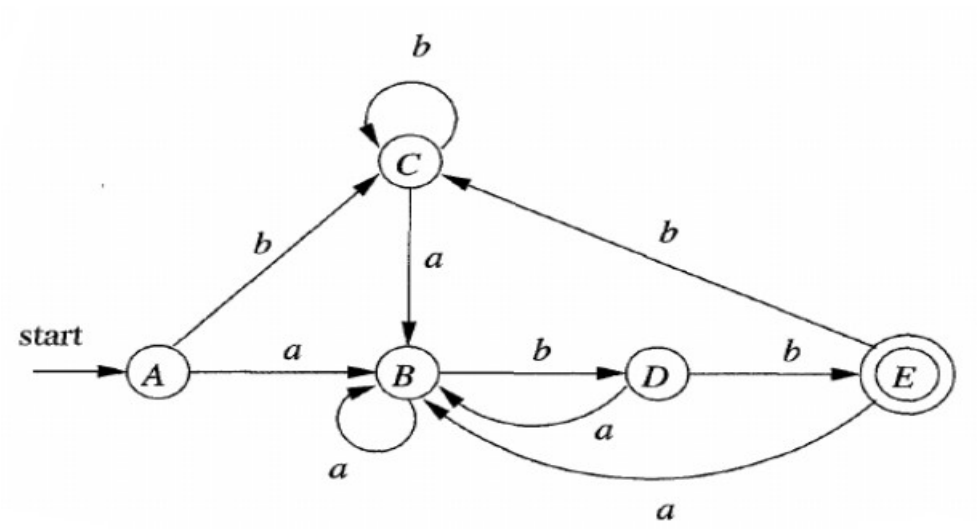


Table de transition de l'AFD :

États de l'AFN	États de l'AFD	a	b
{0,1,2,4,7}	A	B	C
{1,2,3,4,6,7,8}	B	B	D
{1,2,4,5,6,7}	C	B	C
{1,2,4,5,6,7,9}	D	B	E
{1,2,3,5,6,7,10}	E	B	C

6.2.6 Minimisation du nombre des états d'un AFD

La minimisation d'un automate fini déterministe (AFD) est le processus de réduction du nombre d'états de l'AFD tout en maintenant la reconnaissance du même langage. Un AFD minimal est celui qui a le nombre minimal d'états parmi tous les AFD équivalents pour le même langage. La minimisation est utile pour rendre l'automate plus compact et pour faciliter son analyse. Voici un algorithme général pour minimiser un AFD :

Entrée : Un AFD $(Q, \Sigma, \delta, q_0, F)$.

Sortie : Un AFD minimal équivalent $(Q', \Sigma, \delta', q_0', F')$.

a) **Partitionnement des états** :

Divisez l'ensemble d'états Q de l'AFD en deux ensembles initiaux : l'ensemble des états finaux (F) et l'ensemble des états non finaux (Q - F).

Créez une liste de partitions initiale contenant deux ensembles : {F} et {Q - F}.

b) Boucle principale :

Tant que de nouvelles partitions sont créées à chaque itération, faites ce qui suit : a. Sélectionnez une partition P de la liste des partitions. b. Pour chaque symbole σ de l'alphabet Σ, divisez P en sous-partitions en fonction des états accessibles depuis les états de P en suivant les transitions σ. Chaque sous-partition correspondra à un ensemble d'états qui mène à un état dans P avec le symbole σ. c. Si des sous-partitions ont été créées, mettez à jour la liste des partitions avec ces nouvelles partitions.

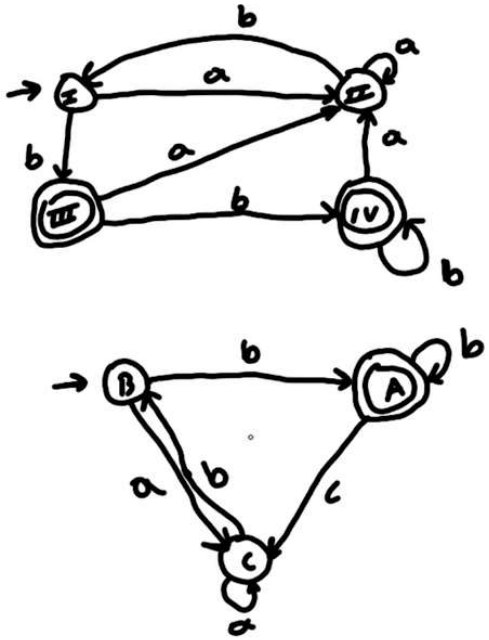
c) Construction de l'AFD minimal :

Une fois que la boucle principale est terminée, vous obtenez une liste de partitions optimale.
Chaque partition dans cette liste représente un état dans l'AFD minimal.
Recalculez les transitions δ' en utilisant les partitions comme états.
L'état initial q_0' sera la partition contenant l'état initial d'origine q_0.
L'ensemble d'états finaux F' sera constitué des partitions qui contiennent des états finaux d'origine.

d) Résultat :

L'AFD résultant est maintenant minimal et équivalent à l'AFD d'origine.

L'algorithme de minimisation d'AFD est basé sur la notion de partitions équivalentes d'états. Il divise les états en groupes tels que les états dans chaque groupe ont le même comportement en termes de transitions et de reconnaissance du langage. Les partitions résultantes représentent les états dans l'AFD minimal. L'algorithme garantit que le nombre d'états dans l'AFD minimal est le plus petit possible tout en préservant la reconnaissance du même langage.



Classe obtenue	Ses états	Les symboles a	Les symboles b
A	II	B	A
	III	B	A
B	I	B	A
	II	B	B

Classe obtenue	Ses états	Les symboles a	Les symboles b
A	III	C	A
	IV	C	A
→ B	I	C	A
C	II	C	B

7. Le générateur d'analyseur lexical Lex/Flex

Lex et Flex sont des générateurs d'analyseurs lexicaux, souvent utilisés pour simplifier la création d'analyseurs lexicaux dans le cadre du développement de compilateurs et d'interprètes. Ces outils automatisent la tâche de conversion des spécifications lexicales en code source pour un analyseur lexical. Voici une présentation générale de Lex/Flex :

Lex/Flex est un outil de génération automatisée d'analyseur lexical, à partir de la spécification des expressions régulières pour décrire les modèles des unités lexicales.

Lex/Flex transforme les modèles (expressions régulières et définitions régulières) en entrée en un diagramme de transition et génére le code source correspondant, dans un fichier appelé lex.yy.c, qui simule le diagramme de transition généré.

Ce fichier écrit en langage C doit être compilé par le compilateur du C pour produire un exécutable qui sera l'analyseur lexical du langage décrit par les modèles en question.

Plusieurs langages dérivés de Lex existent :

- Flex produit du code écrit en C/C++
- Jlex/JFlex produit du code écrit en Java
- Lecl produit du code écrit en Caml
- ml-lex produit du code en ML
- alex produit du code écrit en Haskell
- tcl-tcLex produit du code écrit en tcl

Utilisation de Lex :

a) Un fichier lex.l écrit en langage Lex, qui décrit l'analyseur lexical à générer est présenté au compilateur Lex en entrée.
b) Le compilateur Lex transforme lex.l en un programme écrit en C dans un fichier appelé lex.yy.c.
c) Ce fichier est compilé par le compilateur C et produit un fichier exécutable a.out qui est le fichier du compilateur exécutable, il prendra en paramètre un fichier écrit dans le langage source et produira en sortie une séquence des unités lexicales.

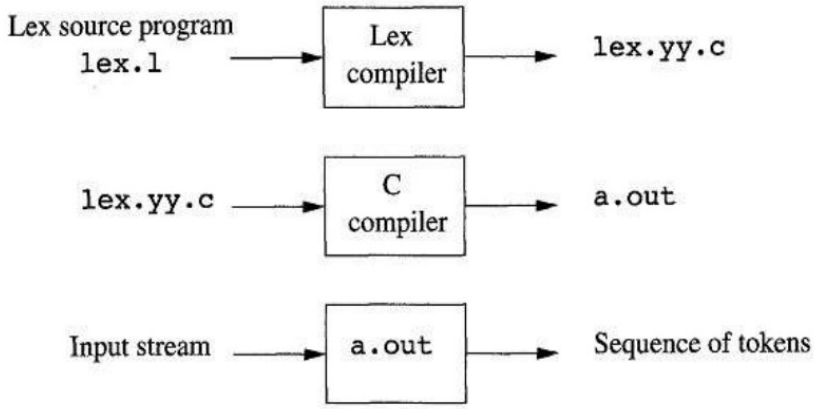


Figure 2.7. Description d'Utilisation de Lex

Un fichier de description pour Lex est formé de trois parties, séparées par des lignes contenant seulement %%, aligné à gauche, selon le schéma suivant : Le générateur d'analyseur lexicale Lex/Flex

```
// Parties déclarations
%{ // (variables, constantes, bibliothèques C, etc.)
%}
//expressions régulières, définitions régulières
%%
// Partie productions (expressions régulières)
%%
// code de service (fonctions utilitaires)
```

- *La partie déclarations* comporte les déclarations des variables, des structures, des unions, des déclarations régulières, des déclaration des bibliothèques en C…

- *La partie productions*, les déclarations prennent la forme suivante :

m_1 {$action_1$} m_2 {$action_2$} ... ... m_i {$action_i$} ... ... m_n {$action_n$}	Les m_i sont des expressions régulières ou des définitions régulières de la partie déclaration. Les $action_i$ sont les actions à réaliser par l'analyseur lexical si un lexème est accepté par m_i. Généralement écrites en langage C, mais d'autres langages peuvent être utilisés.

- ***La 3ème partie*** contient des fonctions qui peuvent être utilisées dans les actions de la 2ème partie.

Eventuellement ces fonctions peuvent être compilées séparément et chargées avec l'analyseur lexical. Cette partie peut aussi contenir la fonction principale main pour une éventuelle exécution du programme produit.

Remarque : Un fichier source en Lex doit avoir l'extension .l (exemple.l). Seulement la partie production est obligatoire dans Lex.

Exemple 1 :

Un programme Lex qui affiche le contenu de son entrée standard vers l'écran.

```
%%   .|\n   ECHO; %%
```

Exemple 2 :

Un programme Lex qui supprime tout les espaces et tabulations.

```
%%   [ \t]    {/* supprime les espaces et tabulations */}
bye {exit(0);}
```

Chapitre 03 :
Analyse Syntaxique

1. Introduction

L'analyse syntaxique est la deuxième étape essentielle du processus de compilation après l'analyse lexicale. Elle consiste à analyser la structure grammaticale d'un programme source pour en déduire la signification ou pour créer une représentation intermédiaire qui peut être utilisée pour la génération de code ultérieure. L'analyse syntaxique vérifie si les séquences de tokens (générées lors de l'analyse lexicale) suivent les règles de syntaxe définies pour le langage source.

2. Grammaires Context-free

Les grammaires contextuelles (ou grammaires context-free en anglais) sont un concept fondamental en théorie de la compilation et en informatique théorique. Elles sont utilisées pour définir formellement la structure syntaxique d'un langage formel, comme un langage de programmation. Une grammaire contextuelle est plus expressive qu'une grammaire régulière (utilisée pour définir des langages réguliers) et moins expressive qu'une grammaire contextuelle sensible (utilisée pour définir des langages contextuels).

2.1. Définition formelle des grammaires

Une grammaire hors contexte comporte quatre composants :

a) Un ensemble de symboles terminaux, parfois appelés « jetons ». Les bornes sont les symboles élémentaires du langage défini par la grammaire.

b) Un ensemble de non-terminaux, parfois appelés « variables syntaxiques ». Chaque non-terminal représente un ensemble de chaînes de terminaux, d'une manière que nous allons décrire.

c) Un ensemble de productions, où chaque production est constituée d'un non-terminal, appelé tête ou côté gauche de la production, d'une flèche et d'une séquence de terminaux et/ou de non-terminaux, appelée corps ou côté droit de la production. L'intention intuitive d'une production est de spécifier l'une des formes écrites d'une construction ; si la tête non terminale représente une construction, alors le corps représente une forme écrite de la construction.

d) Une désignation de l'un des non-terminaux comme symbole de départ. La production est pour un non-terminal si le non-terminal est le responsable de la production. Une chaîne de terminaux est une séquence de zéro ou plusieurs terminaux. La chaîne de bornes zéro, écrite sous la forme E, est appelée chaîne vide.

Les grammaires contextuelles jouent un rôle crucial dans la conception de langages de programmation, la création de compilateurs, et l'analyse syntaxique des programmes. Les outils de génération d'analyseurs syntaxiques, tels que Yacc/Bison pour C/C++ ou ANTLR pour divers langages, sont souvent utilisés pour générer des analyseurs syntaxiques à partir de spécifications de grammaires contextuelles.

Selon la classification de Chomsky, il existe quatre types de grammaires, chacun caractérisé par la forme de ses règles de production. Pour la suite de notre discussion, supposons que V représente un ensemble de symboles (un vocabulaire), alors V* représente l'ensemble de toutes les chaînes de caractères obtenues en concaténant les éléments de V.

Type de grammaire	Forme des règles de production
Type 0 (Grammaires sans restrictions)	$\alpha \rightarrow \beta$ avec $\alpha \in (V_T \cup V_N)^+$ et $\beta \in (V_T \cup V_N)^*$ Autrement dit il n' y a aucune restriction sur α et β
Type 1 (Grammaires à contexte lié ou Contextuelles)	$\alpha \rightarrow \beta$ avec $\alpha \in (V_T \cup V_N)^+$ et $\beta \in (V_T \cup V_N)^*$ et $\lvert\alpha\rvert \leq \lvert\beta\rvert$ et ε (la chaîne vide) ne peut être généré que par l'axiome avec la règle $S \rightarrow \varepsilon$
Type 2 (Grammaires à contexte libre, non contextuelles, algébriques ou encore de Chomsky)	$A \rightarrow \alpha$ avec $A \in V_N$ et $\alpha \in (V_T \cup V_N)^*$
Type 3 (Grammaires régulières ou linéaires droites)	$A \rightarrow \alpha B$ ou $A \rightarrow \alpha$ avec $A, B \in V_N$ et $\alpha \in V_T^*$

Tableau 3.1 : Type de grammaire et la forme des règles de production

2.2 Dérivations et langages

Les dérivations sont un concept important en théorie des langages formels et en analyse syntaxique. Une dérivation est un processus qui montre comment une chaîne de symboles peut être construite en suivant les règles d'une grammaire formelle. Il existe deux types principaux de dérivations : la dérivation gauche (leftmost derivation) et la dérivation droite (rightmost derivation).

Une grammaire dérive des chaînes en commençant par le symbole de début et en remplaçant à plusieurs reprises un non-terminal par le corps d'une production pour ce non-terminal. Les chaînes terminales qui peuvent être dérivées du symbole de début forment le langage défini par la grammaire. Dérivation la plus à gauche et la plus à droite d'une chaîne

• ***Dérivation la plus à gauche*** - Une dérivation la plus à gauche est obtenue en appliquant la production à la variable la plus à gauche à chaque étape. Les deux dérivations les plus à gauche pour une chaîne d'entrée donnée sont :

E => E +E
=> id + E
=> id + E * E
=> id + id * E
=> id + id * id
(a)

E => E * E
=> E+E*E
=> id + E * E
=> id+ id* E
=> id + id * id
(b)

• ***Dérivation la plus à droite*** - Une dérivation la plus à droite est obtenue en appliquant la production à la variable la plus à droite à chaque étape.

Exemple : Supposons que n'importe quel ensemble de règles de production dans un CFG soit : X → X+X | X*X |X| a sur un alphabet {a}.

La dérivation la plus à gauche de la chaîne "a+a*a" est :

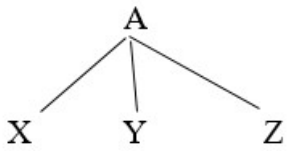
X → X+X → a+X → a + X*X → a+a*X → a+a*a

La dérivation la plus à droite pour la chaîne ci-dessus "a+a*a" est :

X → X*X → X*a → X+X*a → X+a*a → a+a*a

2.3 Dérivation ou rendement d'un arbre

La dérivation ou le rendement d'un arbre d'analyse est la chaîne finale obtenue en concaténant les étiquettes des feuilles de l'arbre de gauche à droite, en ignorant les valeurs Nulle. Cependant, si toutes les feuilles sont Nulles, la dérivation est Nulle. L'arbre d'analyse montre de manière imagée comment le symbole de début d'une grammaire dérive une chaîne dans la langue. Si le non-terminal A a une production A → XYZ, alors un arbre d'analyse peut avoir un nœud intérieur étiqueté A avec trois enfants étiquetés X, Y et Z, de gauche à droite :



Étant donné une grammaire hors contexte, un arbre d'analyse selon la grammaire est un arbre avec les propriétés suivantes :

1. La racine est étiquetée par le symbole de départ.
2. Chaque feuille est étiquetée par un terminal ou par e.
3. Chaque nœud intérieur est étiqueté par un non-terminal

Si A est l'étiquetage non terminal d'un nœud intérieur et Xi, Xz, . . . , Xn sont les étiquettes des enfants de ce nœud de gauche à droite, alors il doit y avoir une production A → X1X2... Xn. Ici, X1, X2,. . . , Xn, chacun représente un symbole qui est soit un terminal, soit un non-terminal. Comme cas particulier, si (A → c) est une production, alors un nœud étiqueté A peut avoir un seul enfant étiqueté E.

3. Arbres syntaxiques

Un arbre d'analyse (parfois appelé arbre syntaxique) est une structure de données arborescente qui représente la structure syntaxique d'une phrase ou d'une expression conformément à une grammaire formelle. Les arbres d'analyse sont couramment utilisés en analyse syntaxique pour représenter la manière dont les règles grammaticales d'un langage génèrent une phrase ou une expression. Ils jouent un rôle essentiel dans la compilation, l'interprétation et la compréhension des langages de programmation et des langages naturels.

Caractéristiques principales d'un arbre d'analyse :

a) **Racine de l'arbre** : L'arbre d'analyse a une racine qui représente la phrase ou l'expression globale.

b) **Nœuds** : Les nœuds de l'arbre représentent des éléments grammaticaux tels que des symboles non terminaux, des opérations ou des mots. Chaque nœud est étiqueté avec le symbole ou le mot qu'il représente.

c) **Arêtes** : Les arêtes entre les nœuds représentent les relations de dérivation ou les transitions entre les éléments grammaticaux. Par exemple, une arête peut représenter une règle de production dans une grammaire formelle.

d) **Feuilles de l'arbre** : Les nœuds de l'arbre qui n'ont pas d'enfants sont appelés les feuilles de l'arbre. Les feuilles correspondent généralement aux symboles terminaux, c'est-à-dire aux éléments concrets du langage tels que les mots dans une phrase ou les tokens dans un programme informatique.

e) **Branches et sous-arbres** : Les sous-arbres de l'arbre d'analyse représentent des sous-phrases ou des sous-expressions de la phrase ou de l'expression globale. Chaque branche de l'arbre correspond à l'application d'une règle grammaticale.

f) **Structure hiérarchique** : Les arbres d'analyse sont hiérarchiques, ce qui signifie que chaque nœud parent englobe ses nœuds enfants, reflétant la structure imbriquée des éléments grammaticaux dans la phrase ou l'expression.

g) **Exemple d'arbre d'analyse** : Voici un exemple simple d'arbre d'analyse pour l'expression arithmétique "- (id + id) " basée sur une grammaire d'expression arithmétique :

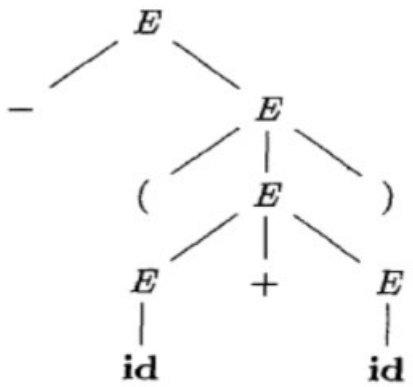


Figure 3.1. Arbre d'analyse pour l'expression arithmétique "- (id + id) "

La figure suivante montre la construction étape par étape de l'arborescence d'analyse à l'aide de CFG pour l'arborescence d'analyse pour la chaîne d'entrée - (id + id).

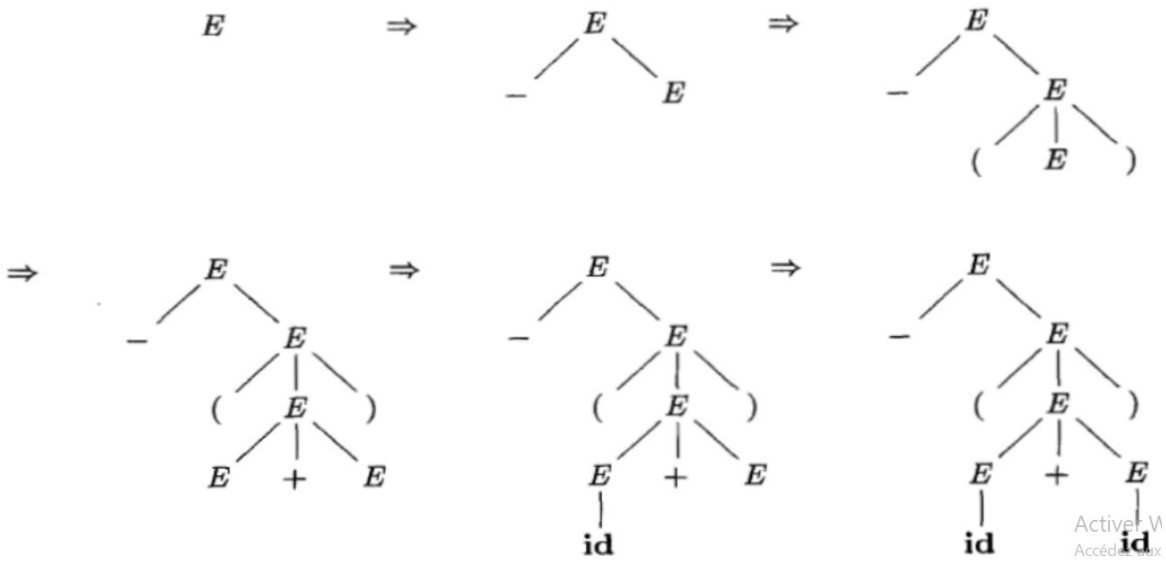


Figure 3.2. la construction étape par étape de l'arborescence d'analyse "- (id + id) "

Exemple

L'arbre d'analyse pour la chaîne d'entrée id+id*id en utilisant la grammaire sans contexte ci-dessus est :

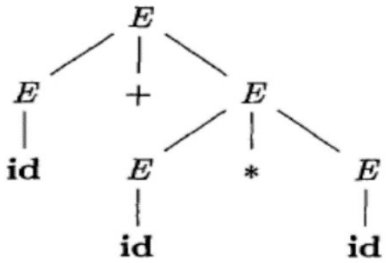


Figure 3.3. Arbre d'analyse pour l'expression arithmétique " id+id*id "

4. Ambiguïté

L'ambiguïté est un concept important en théorie des langages formels et en analyse syntaxique. Elle se produit lorsqu'une phrase ou une expression dans un langage peut être analysée de plusieurs manières différentes selon les règles de grammaire du langage. En d'autres termes, une construction grammaticale peut avoir plusieurs arbres d'analyse ou plusieurs interprétations valides.

Voici un exemple simple pour illustrer l'ambiguïté en utilisant une grammaire pour les expressions arithmétiques :

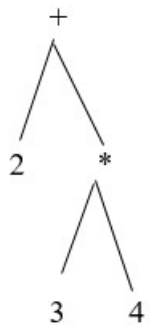
Supposons la grammaire suivante :

expr-> expr+expr | expr*expr | (expr) | Num

Avec cette grammaire, l'expression "2 + 3 * 4" peut être analysée de deux manières différentes:

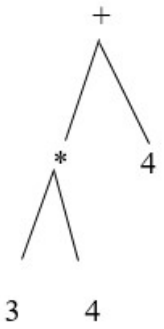
- Analyse gauche (leftmost) :

Dans cette analyse, l'addition est effectuée avant la multiplication :



- Analyse droite (rightmost) :

Dans cette analyse, la multiplication est effectuée avant l'addition :



Dans cet exemple, la phrase "2 + 3 * 4" est ambiguë car elle peut être interprétée de deux manières différentes, selon la manière dont on choisit de suivre les règles de grammaire pour les opérations d'addition et de multiplication.

L'ambiguïté peut poser des problèmes dans l'analyse syntaxique et la sémantique d'un langage, car elle peut rendre difficile la détermination de la signification précise d'une construction grammaticale. Pour résoudre l'ambiguïté, il est souvent nécessaire de clarifier les règles de grammaire en utilisant des priorités d'opérations ou des règles spécifiques pour gérer les cas ambigus. Dans le cas des langages de programmation, la spécification du langage définit généralement comment les constructions ambiguës doivent être interprétées pour garantir une sémantique cohérente.

5. Élimination de l'ambigüité

L'élimination de l'ambiguïté dans une grammaire est une étape cruciale dans la conception de langages formels et dans la construction de compilateurs. L'ambiguïté peut entraîner des interprétations ambiguës et des problèmes de compréhension dans le traitement des programmes ou des langages naturels. Voici quelques techniques couramment utilisées pour éliminer l'ambiguïté dans les grammaires :

5.1 Éliminer la récursion à gauche

L'élimination de la récursion à gauche est une technique courante utilisée pour rendre les grammaires non ambiguës et adaptées à l'analyse ascendante (bottom-up). La récursion à gauche se produit lorsqu'une règle de production permet à une variable (symbole non terminal) de se dériver directement en elle-même ou en une séquence qui finit par se réduire en elle-même. Cela peut entraîner des ambiguïtés lors de l'analyse syntaxique.

Voici un exemple de récursion à gauche dans une grammaire simple :

Parce que nous essayons de générer une dérivation la plus à gauche en balayant l'entrée de gauche à droite, les grammaires de la forme $A \rightarrow A\ x$ peuvent provoquer une récursion sans fin. De telles grammaires sont dites récursives à gauche et elles doivent être transformées si nous voulons utiliser un analyseur descendant.

• Une grammaire est laissée récursive si, pour un A non terminal, il existe une dérivation $A \Rightarrow^+ A\alpha$

• Pour éliminer la récursion directe à gauche, remplacez

1) $A \rightarrow A\alpha \mid \beta$ avec $A' \rightarrow \alpha\ A' \mid \varepsilon$

2) $A \rightarrow A\alpha 1 \mid A\alpha 2 \mid ... \mid A\alpha m \mid \beta 1 \mid \beta 2 \mid ... \mid \beta n$ avec $A \rightarrow \beta 1B \mid \beta 2B \mid ... \mid \beta nB$

$B \rightarrow \alpha 1B \mid \alpha 2B \mid ... \mid \alpha mB \mid \varepsilon$

5.2. Factorisation à gauche

La factorisation à gauche permet de rendre la grammaire non ambiguë et adaptée à l'analyse syntaxique sans ambiguïté. Cette technique est couramment utilisée dans la conception de langages formels et dans la construction de compilateurs pour garantir une analyse syntaxique correcte et efficace.

La factorisation à gauche est une transformation grammaticale utile pour produire une grammaire adaptée à l'analyse prédictive. Lorsqu'il n'est pas clair laquelle des deux productions alternatives utiliser pour développer une production A non-terminale, nous pouvons réécrire les

productions A pour différer la décision jusqu'à ce que nous ayons vu suffisamment d'informations pour faire le bon choix.

• Considérons S → si E alors S sinon S | si E alors S

• Laquelle des deux productions devrions-nous utiliser pour étendre le S non-terminal lorsque le prochain jeton est if ?

Nous pouvons résoudre ce problème en excluant la partie commune de ces règles.

A → αβ1 | αβ2 |...| αβn | γ devient A → αB| γ B → β1 | β2 |...| βn

Considérons la grammaire, G :

S → iEtS | iEtSeS | a

E → b

Factorisée à gauche, cette grammaire devient

S → iEtSS' | a

S’ → eS |ε

E → b

Chapitre 04 :
Analyse Syntaxique Descendante

1. Analyse descendante

L'analyse syntaxique descendante (ou top-down parsing en anglais) est l'un des deux principaux types d'analyse syntaxique utilisés dans le processus de compilation des langages de programmation. Il s'agit d'une méthode qui vise à construire un arbre syntaxique (ou une autre forme de structure de données) à partir du haut de la hiérarchie grammaticale, en partant du symbole de départ (axiome) de la grammaire et en descendant vers les feuilles de l'arbre syntaxique, qui représentent les symboles terminaux (tokens) du langage.

Il existe deux variantes de cet analyseur, comme indiqué ci-dessous.

a. Analyse de descente récursive

b. Analyse prédictive pilotée par table : analyse LL (1)

2. Analyse de descente récursive

Un programme d'analyse par descente récursive se compose d'un ensemble de procédures récursives, une pour chaque non-terminal. Chaque procédure est responsable de l'analyse des constructions définies par son non-terminal. L'exécution commence par la procédure pour le symbole de départ, qui s'arrête et annonce le succès si son corps de procédure analyse la totalité de la chaîne d'entrée. Si la grammaire donnée est :

E → TE′
E′ → +TE′ | €
T → FT′
T′ → *FT′ | €
F → (E) | id

Les procédures récursives pour l'analyseur de descente récursive pour la grammaire donnée sont données ci-dessous.

```
procedure E( ) {
     T( );
     E′( ); }

Procedure E′( ) {
   if input = ‘+‘ {
      advance( );
      T ( );
      E′( );
    return true; }
 else error; }


procedure F( ) {
     if input = ‘(‘ {
        advance( );
        E ( );
       if input = ‘)‘ advance( );
       return true; }
       else if input = ‘id’ {
          advance( ); return true; }
    else return error; }
```

```
procedure T ( ) {
    F( );
    T′( ); }

procedure T′( ) {
   if input = ‘*‘ {
       advance( );
       F ( );
       T′( );
       return true; }
  else return error; }


   advance() {
       input = next token;}
```

3. Analyse prédictive

L'analyse prédictive est une méthode d'analyse syntaxique utilisée dans le processus de compilation des langages de programmation. C'est une forme d'analyse syntaxique descendante qui consiste à prédire la prochaine étape de l'analyse en se basant uniquement sur le symbole courant du flux d'entrée, sans avoir besoin de regarder plusieurs symboles à l'avance. L'analyse prédictive est souvent basée sur des grammaires LL(k), où "LL" signifie "Left-to-Right, Leftmost derivation" (de gauche à droite, dérivation la plus à gauche) et "k" représente le nombre de symboles anticipés nécessaires pour prendre une décision, il possède les éléments représentés par la figure 4.1.

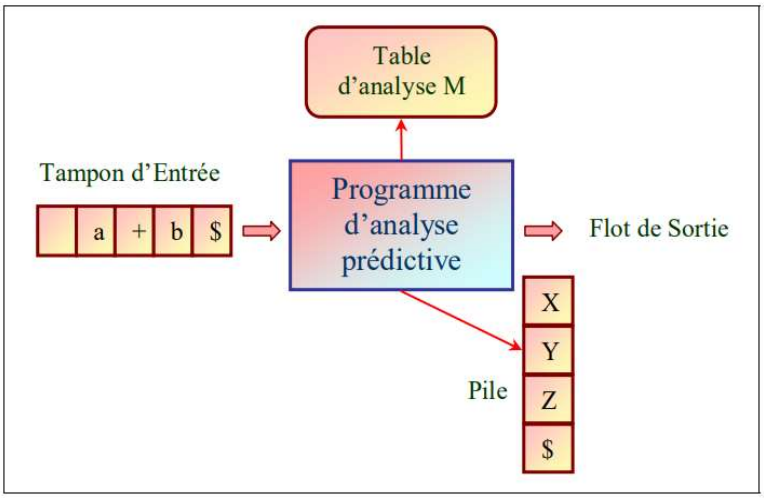


Figure 4.1. Modèle d'analyseur prédictif non récursif

Un analyseur prédictif piloté par table possède un tampon d'entrée, une pile, une table d'analyse et un flux de sortie. Le tampon d'entrée contient la chaîne à analyser, suivie de $, un symbole utilisé comme marqueur de fin droit pour indiquer la fin de la chaîne d'entrée. La pile contient une séquence de symboles de grammaire avec $ en bas, indiquant le bas de la pile. Initialement, la pile contient le symbole de début de la grammaire au-dessus de $. La table d'analyse est un tableau à deux dimensions M[A,a] où A est un non-terminal et a est un terminal ou le symbole $. L'analyseur est contrôlé par un programme qui se comporte comme suit. Le programme considère X, le symbole en haut de la pile, et a, le symbole d'entrée actuel. Ces deux symboles déterminent l'action de l'analyseur. Il y a trois possibilités.

1. Si X= a=$, l'analyseur s'arrête et annonce la réussite de l'analyse.
2. Si X=a!=$, l'analyseur retire X de la pile et avance le pointeur d'entrée jusqu'au symbole d'entrée suivant.
3. Si X est un non-terminal, le programme consulte l'entrée M[X,a] de la table d'analyse M. Cette entrée sera soit une production X de la grammaire, soit une entrée d'erreur.

Si, par exemple, M[X, a]={X- >UVW}, l'analyseur remplace X en haut de la pile par WVU (avec U en haut). En sortie, nous supposerons que l'analyseur imprime simplement la production utilisée ; n'importe quel autre code pourrait être exécuté ici.

Si M[X, a]=erreur, l'analyseur appelle une routine de récupération d'erreur.

3.1 Fonctions Premier et Suivant

Les fonctions Premier (First) et Suivant (Follow) sont des concepts clés en théorie des langages formels et en analyse syntaxique. Elles sont largement utilisées pour analyser et générer des arbres syntaxiques à partir de grammaires formelles. Ces fonctions aident à

déterminer les premiers symboles (Premier) qui peuvent apparaître dans une chaîne dérivée à partir d'un symbole non terminal donné et les symboles qui peuvent suivre (Suivant) ce symbole non terminal dans une dérivation.

3.1.1 Fonction Premier (Début ou First)

La fonction Premier est essentielle en analyse syntaxique pour déterminer les choix possibles lors de la construction de l'arbre syntaxique. Elle aide à décider quelle règle de production appliquer en fonction du symbole courant du flux d'entrée.

La fonction FIRST calcule l'ensemble des symboles terminaux par lesquels commence la partie droite des productions. Pour calculer PREMIER (A) pour tous les symboles de grammaire, appliquez les règles suivantes jusqu'à ce qu'aucun terminal ou ε ne puisse plus être ajouté à un premier ensemble.

Règle 1. Si X est terminal, alors FIRST(X) est {X}.

Règle 2. Si X → ε est une production, alors ajoutez ε à FIRST(X).

Règle 3. Si X est non terminal et X → aα est une production alors ajoutez a à FIRST(X).

Règle 4. Si X est non terminal et X → Y1 Y2…Yk est une production, alors placez a dans FIRST(X) si pour certains i, a est dans FIRST(Yi), et ε est dans tout PREMIER(Y1),…, PREMIER(Yi-1); c'est-à-dire Y1,….Yi-1=> ε. Si ε est dans FIRST(Yj) pour tout j=1,2,..,k, alors ajoutez ε à FIRST(X).

Exemple 1

Soient les règles de productions suivantes pour une grammaire donnée :

S → Ba

B → cP | bP | P| ε

P → dS

On se propose de déterminer les premiers du symbole S. On a :

S → Ba → cPa donc: c ∈ Prem(S)

On a: S → Ba → bPa donc: b ∈ Prem(S)

On a: S → Ba → Pa → dSa donc: d ∈ Prem(S)

On a: S → Ba → εa → a donc: a ∈ Prem(S)

Nous pouvons en déduire que Prem(S)= {a, b, c, d}

Exemple 2

Soient les règles de productions suivantes pour une grammaire donnée :

S → ABCe

A → aA | ε

B → bB | cB | ε

C → de | da | dA

On se propose de déterminer les premiers de S, A B et C en appliquant les règles énoncées précédemment :

Prem(C) = {d}

Prem(B) = {b, c, ε}

Prem(A) = {a, ε}

Prem(S) = {a} U {b, c} U {d} = {a, b, c, d}

3.1.2 Fonction Suivant (Follow)

Follow (A) n'est rien d'autre que l'ensemble des symboles terminaux de la grammaire qui suivent immédiatement le non-terminal A. Si a est immédiatement à droite du non-terminal A, alors Follow(A)= {a}. Pour calculer FOLLOW (A) pour tous les non-terminaux A, appliquez les règles suivantes jusqu'à ce qu'aucun symbole ne puisse plus être ajouté à un ensemble FOLLOW.

Règle 1. Si S est un symbole de début, alors FOLLOW(S) contient $.

Règle 2. S'il y a une production A → αBβ, alors tout dans FIRST(β) sauf ε est placé dans suivre(B).

Règle 3. S'il existe une production A → αB, ou une production A → αBβ où PREMIER(β) contient ε, alors tout dans FOLLOW(A) est dans FOLLOW(B).

Exemple 3

Soient les règles de productions suivantes pour une grammaire donnée :

S → aAB | aAd

B → bB| c

On se propose de déterminer les suivants du symbole A. On constate, d'après la première règle de production, que A peut être suivi par B ou d. D'après la deuxième règle, B peut commencer par b ou c. On peut donc déduire que : Suiv(A)= {b, c, d}

Exemple 4

Calculer les valeurs FIRST et FOLLOW de la grammaire de l'expression

1. **E → TE′**
2. **E′→+TE′ | €**
3. **T → FT′**
4. **T′ →*FT′ | €**
5. **F → (E) | id**

Calcul les valeurs des PREMIÈRES :

FIRST (E) = FIRST (T) = FIRST (F) = {(, id}

FIRST (E′) = {+, €}

FIRST (T′) = {*, €}

Calcul les valeurs des SUIVANTS :

FOLLOW (E) = { $,)} Parce que c'est le symbole de début de la grammaire.

FOLLOW (E′) = {FOLLOW (E)} satisfaisant la 3ème règle de FOLLOW()

= { $,) }

FOLLOW (T) = { PREMIER E′}U{ FOLLOW (E′) } Il satisfait à la 2ème règle.

={+, FOLLOW (E′)}

= { +, $,) }

FOLLOW (T′) = { FOLLOW (T)} Satisfaire la 3ème règle

= { +, $,) }

FOLLOW (F) = { PREMIER (T′)} U { FOLLOW (E′) } Il satisfait à la 2ème règle.

= {*, FOLLOW (T)}

= { *, +, $,)}

NON TERMINAL	FIRST	FOLLOW
E	{ (, id }	{ $,) }
E′	{ +, € }	{ $,) }
T	{ (, id}	{ +, $,) }
T′	{*, €}	{ +, $,) }
F	{ (, id}	{ *, +, $,) }

Tableau 4.1 : FIRST and FOLLOW values

3.2. Construction d'une table d'analyse prédictive ou LL (1)

La construction d'une table d'analyse prédictive, souvent appelée table LL(1), est une étape essentielle dans la mise en œuvre d'un analyseur syntaxique prédictif pour un langage de programmation donné. La table LL(1) est utilisée pour prendre des décisions sur laquelle des règles de production de la grammaire doit être appliquée lors de l'analyse syntaxique, en fonction du symbole courant dans le flux d'entrée et du sommet de la pile (en haut de la pile d'analyse).

Ce processus consistant à placer toutes les productions de la grammaire dans la table d'analyse basée sur les valeurs FIRST et FOLLOW des productions. Les règles à suivre pour construire la table d'analyse (M) sont :

1. Pour chaque production A-> α de la grammaire, suivez les étapes ci-dessous.
2. Pour chaque symbole terminal 'a' dans FIRST (α), ajoutez la production A-> α à M [A, a].
3. i. Si ε est dans FIRST (α), ajoutez la production A->α à M [A, b], où b représente tous les terminaux de SUIVEZ (A).
   ii. Si ε est dans FIRST(α) et $ est dans FOLLOW (A) alors ajoutez la production A->α à M [A, $]. 4. Marquez les autres entrées de la table d'analyse comme erreurs.

	+	*	(	)	id	$
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→id	

Tableau 4.2 : LL (1) Table d'analyse pour la grammaire des expressions

Remarque : s'il n'y a pas plusieurs entrées dans le tableau pour un seul terminal, la grammaire est acceptée par l'analyseur LL(1).

3.3. Algorithme d'analyse LL (1)

L'analyseur agit sur la base de deux symboles je. A, le symbole en haut de la pile ii. a, le symbole d'entrée actuel Il existe trois conditions pour A et 'a', qui sont utilisées pour le programme d'analyse.

1. Si A=a=$ alors l'analyse est réussie.
2. Si A=a≠$ alors l'analyseur sort de la pile et avance le pointeur d'entrée actuel vers le suivant.
3. Si A est un non-terminal, l'analyseur consulte l'entrée M [A, a] dans la table d'analyse.

Si M[A, a] est une Production A-> X1X2..Xn, alors le programme remplace le A en haut de la Stack par X1X2..Xn de telle sorte que X1 vienne en haut.

Acceptation des chaînes par l'analyseur

Si la chaîne d'entrée de l'analyseur est id + id * id, le tableau ci-dessous montre comment l'analyseur accepte la chaîne à l'aide de Stack.

Stack	Input	Action	Comments
$E	id+ id* id $	E TE`	E on top of the stack is replaced by TE`
$E`T	id+ id*id $	T FT`	T on top of the stack is replaced by FT`
$E`T`F	id+ id*id $	F id	F on top of the stack is replaced by id
$E`T`id	id+ id*id $	pop and remove id	Condition 2 is satisfied
$E`T`	+id*id$	T €	T` on top of the stack is replaced by €
$E`	+id*id$	E` +TE`	E` on top of the stack is replaced by +TE`
$E`T+	+id*id$	Pop and remove +	Condition 2 is satisfied
$E`T	id*id$	T FT`	T on top of the stack is replaced by FT`
$E`T`F	id*id$	F id	F on top of the stack is replaced by id
$E`T`id	id * id$	pop and remove id	Condition 2 is satisfied
$E`T`	*id$	T` *FT`	T` on top of the stack is replaced by *FT`
$E`T`F*	*id$	pop and remove *	Condition 2 is satisfied
$E`T`F	id$	F id	F on top of the stack is replaced by id
$E`T`id	id$	Pop and remove id	Condition 2 is satisfied
$E`T`	$	T` €	T` on top of the stack is replaced by €
$E`	$	E` €	E` on top of the stack is replaced by €
$	$	Parsing is successful	Condition 1 satisfied

Tableau 4.3 : Séquence des étapes prises par l'analyseur lors de l'analyse du flux de jetons d'entrée id+ id* id

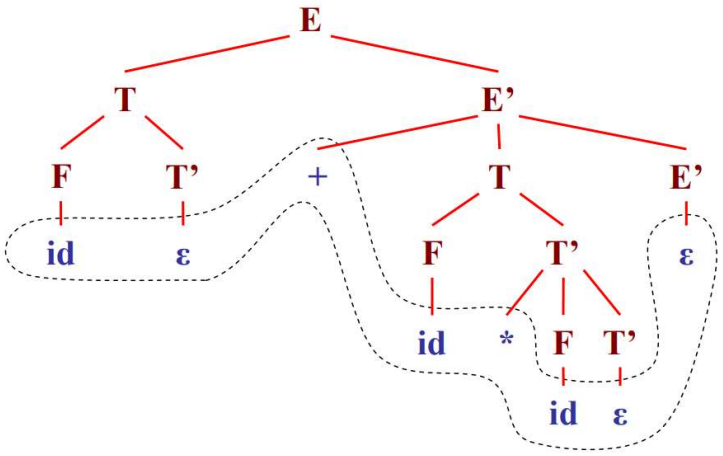


Figure 4.2 : Arbre d'analyse pour l'identifiant d'entrée + id* id

3.4 Conditions pour qu'une grammaire soit LL(1)

On peut montrer formellement qu'une grammaire est LL(1), si et seulement si, à chaque fois que l'on a une production de la forme A→ α|β, les trois conditions suivantes sont vérifiées :

Condition 1 : Pour aucun terminal a, α et β ne se dérivent toutes les deux en des chaînes commençant par a. Autrement dit : Prem(α) ∩ Prem(β) = ∅.

Condition 2 : Une des chaînes (α ou β) au plus peut se dériver en la chaîne vide. Autrement dit : (β→* ε) ou bien (α→* ε) mais pas les deux à la fois.

Condition 3 : Si (β→* ε), α ne se dérive pas en une chaîne commençant par un terminal de Suiv(A). Autrement dit : Si (β→* ε) alors Prem(α) ∩ Suiv(A) = ∅.

Exemple 5

On se propose de montrer si la grammaire G1 ayant les règles de productions suivantes est LL(1).

S → aAb

A → cd | c

Pour la production A → cd | c, on constate que Prem(cd) ∩ Prem(c) = {c} ∩ {c} = {c} ≠ ∅ , ceci signifie que la condition C1 n'est pas vérifiée donc la grammaire G1 n'est pas LL(1).

Exemple 6

Soient les règles de productions suivantes pour une grammaire G2 dont l'axiome est:

E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | id

On se propose de montrer si G2 est une grammaire LL(1).

Pour la production E' → +TE' | ε

C1 : Prem(+TE') ∩ Prem(ε) = {+} ∩ {ε} = ∅ vérifiée

C2 : vérifiée car +TE' ne peut pas se dériver en ε

C3 : Prem(+TE') ∩ Suiv(E') = {+} ∩ {$,)} = ∅ vérifiée

Pour la production T' → *FT' | ε

C1 : Prem(*FT') ∩ Prem(ε) = {*} ∩ {ε} = ∅ vérifiée

C2 : vérifiée car *FT' ne peut pas se dériver en ε

C3 : Prem(*FT') ∩ Suiv(T') = {*} ∩ {+, $,)} = ∅ vérifiée

Pour la production F → (E) | id

C1 : Prem((E)) ∩ Prem(id) = {(} ∩ {id} = ∅ vérifiée

C2 : vérifiée car (E) →* ε et id→* ε sont toutes les deux impossibles

C3 : non applicable (à cause de C2) vérifiée

Conclusion : Les trois conditions (C1, C2 et C3) sont vérifiées pour toutes les productions de la forme A→α|β, donc la grammaire G2 est LL(1).

3.5. Gestion des erreurs (récupération) dans l'analyse prédictive

La pile d'un analyseur prédictif non récursif rend explicites les terminaux et les non-terminaux que l'analyseur espère faire correspondre avec le reste de l'entrée. Nous ferons donc référence aux symboles sur la pile de l'analyseur dans la discussion suivante. Une erreur est détectée lors de l'analyse prédictive lorsque le terminal au sommet de la pile ne correspond pas au symbole d'entrée suivant ou lorsque le non-terminal A est au sommet de la pile, a est le symbole d'entrée suivant et l'entrée de la table d'analyse M[A,a] est vide. Envisagez l'analyse prédictive de récupération d'erreur à l'aide des deux méthodes suivantes

- *Récupération en mode panique*
- *Récupération du niveau de phrase*

- ***Récupération d'erreur en mode panique***

est basé sur l'idée de sauter des symboles sur l'entrée jusqu'à ce qu'un jeton dans un ensemble sélectionné de jetons de synchronisation apparaisse. Son efficacité dépend du choix du kit de synchronisation.

Les ensembles doivent être choisis de manière à ce que l'analyseur récupère rapidement des erreurs susceptibles de se produire dans la pratique. Comme point de départ, nous pouvons placer tous les symboles de FOLLOW(A) dans l'ensemble de synchronisation pour le non-terminal A. Si nous sautons les jetons jusqu'à ce qu'un élément de FOLLOW(A) soit vu et que nous retirons A de la pile, il est probable que l'analyse puisse continuer. Il ne suffit pas d'utiliser FOLLOW(A) comme ensemble de synchronisation pour A.

Par exemple, si des points-virgules terminent des instructions, comme en C, alors les mots-clés qui commencent les instructions peuvent ne pas apparaître dans l'ensemble FOLLOW des expressions génératrices non-terminales. Un point-virgule manquant après une affectation peut donc entraîner l'omission du mot-clé commençant par l'instruction suivante. Il existe souvent une structure hiérarchique sur les constructions dans un langage ; par exemple, les expressions apparaissent dans l'instruction, qui apparaissent dans les blocs, et ainsi de suite. Nous pouvons ajouter à l'ensemble de synchronisation d'une construction inférieure les symboles qui commencent les constructions supérieures. Par exemple, nous pourrions ajouter des mots-clés qui commencent les instructions aux ensembles de synchronisation pour les expressions générées par les non-terminaux.

Si nous ajoutons des symboles dans FIRST(A) à l'ensemble de synchronisation pour le non-terminal A, alors il peut être possible de reprendre l'analyse selon A si un symbole dans FIRST(A) apparaît dans l'entrée.

Si un non-terminal peut générer la chaîne vide, alors la production dérivant e peut être utilisée par défaut. Cela peut retarder la détection d'une erreur, mais ne peut pas faire manquer une erreur. Cette approche réduit le nombre de non-terminaux à prendre en compte lors de la récupération d'erreur.

Si un terminal au sommet de la pile ne peut pas être mis en correspondance, une idée simple consiste à faire apparaître le terminal, à émettre un message indiquant que le terminal a été inséré et à poursuivre l'analyse. En effet, cette approche considère que l'ensemble de synchronisation d'un jeton est constitué de tous les autres jetons.

- Récupération du niveau de phrase

Cela implique de définir les entrées vides dans le tableau avec des pointeurs vers certaines routines d'erreur qui peuvent modifier, supprimer ou insérer des symboles dans l'entrée ou peuvent également faire apparaître des symboles de la pile.

Chapitre 05 :
Analyse Syntaxique Ascendante

1. Analyse Ascendante

L'analyse ascendante est une méthode essentielle dans le processus de compilation qui permet de déterminer la structure syntaxique d'un programme en commençant par les tokens et en construisant progressivement l'arbre syntaxique. Elle est utilisée pour vérifier la syntaxe correcte du code source et pour effectuer d'autres analyses nécessaires dans le processus de compilation.

L'analyse ascendante correspond à la construction d'un arbre d'analyse pour une chaîne d'entrée commençant par les feuilles (les nœuds inférieurs) et remontant vers la racine (le nœud supérieur). Cela implique de « réduire une chaîne d'entrée 'w' au symbole de début de la grammaire. À chaque étape de réduction, une sous-chaîne particulière correspondant au côté droit de la production est remplacée par un symbole à gauche de cette production et c'est la dérivation la plus à droite.

Par exemple, considérons la grammaire suivante :

E → E+T|T

T → T*F

F → (E)|id

L'analyse ascendante de la chaîne d'entrée « id * id » est la suivante :

INPUT STRING	SUB STRING	REDUCING PRODUCTION
id*id	Id	F->id
F*id	T	F->T
T*id	Id	F->id
T*F	*	T->T*F
T	T*F	E->T
E		Start symbol. Hence, the input String is accepted

Tableau 5.1 : Analyse ascendante de la chaîne d'entrée « id * id »

L'analyse ascendante est classée en :

1. Analyse décalage-réduction (Shift-Reduce),
2. Analyse de la priorité des opérateurs, et
3. Tables d'analyse LR:
    - 3.i. SLR(1)
    - 3.ii. CALR (1)
    - 3.iii.LALR(1)

2. Analyse Shift-Reduce

production dont la partie droite est ni d'avoir deux non terminaux adjacents. Une grammaire vérifiant la dernière propriété est une grammaire d'opérateurs.

Dans l'analyse par précédence d'opérateurs, nous définissons trois relations de précédence disjointes < . ,= . , . >, entre certains couples de terminaux.

A< . b : a prend la précédence sur b

A=. b : a la même précédence que b

a >. b : a prend la précédence sur b

Habituellement, il y'a deux façons de déterminer quelles relation de précédence doivent exister entre les couples de terminaux, la première méthode est intuitive, elle est fondée sur les notions traditionnelles d'associativité et de priorité d'opérateurs. Par exemple, si * a une priorité plus grande que +, on a +<.* et *.>+.

La seconde méthode consiste à construire d'abord une grammaire non ambiguë pour le langage qui reflète l'associativité et la priorité correcte dans ses arbres d'analyse, il existe une méthode mécanique pour en dériver les relations de précédence d'opérateurs.

Par exemple considérons la prot-phrase droite id+id*id et supposons que les relations de précédence sont les suivantes :

	Id	+	*	#
Id		.>	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
#	<.	<.	<.	

Tableau 5.2: Analyse Shift-Reduce, les relations de précédence « id+id*id »

Alors, la chaine dans laquelle les relations de précédence ont été insérées est : # + * #, on peut déterminer le manche en appliquant le procédé suivant : 1. Parcourir la chaine à partir de l'extrémité gauche jusqu'à rencontrer le premier.>, dans notre exemple cela se produit entre le premier id et +. 2. Parcourir alors en sens inverse (vers la gauche) en sautant les =. Jusqu'à rencontrer un et à droite du # qui indique que l'extrémité gauche du manche se trouve entre + et * et que son extrémité droite se trouve entre * et #, alors le manche est E*E.

3. Analyse de la priorité des opérateurs

La grammaire de priorité des opérateurs est une sorte de méthode d'analyse par réduction de décalage qui peut être appliquée à une petite classe de grammaires d'opérateurs. Et il peut également traiter des grammaires ambiguës.

• Une grammaire d'opérateur a deux caractéristiques importantes :

1. Il n'y a pas de productions ε.
2. Aucune production n'aurait deux non-terminaux adjacents.

• La grammaire des opérateurs pour accepter les expressions est donnée ci-dessous :

E ➡ E+E / E → E-E / E → E*E / E → E/E / E → E^E / E → -E / E → (E) / E → id

Les deux principaux défis liés à l'analyse de la priorité des opérateurs sont :

1. Identification des poignées correctes dans l'étape de réduction, de telle sorte que l'entrée donnée doit être réduite au symbole de départ de la grammaire.

2. Identification de la production à utiliser pour la réduction dans les étapes de réduction, de telle sorte que nous devrions correctement réduire l'entrée donnée au symbole de départ de la grammaire.

L'analyseur de priorité des opérateurs se compose de :

1. Un tampon d'entrée qui contient une chaîne à analyser suivie de a$, un symbole utilisé pour indiquer la fin de l'entrée.

2. Une pile contenant une séquence de symboles de grammaire avec un $ en bas de la pile.

3. Une table de relations de préséance des opérateurs O, contenant les relations de préséance entre la paire de terminaux. Il existe trois types de relations de préséance entre la paire de paires de terminaux « a » et « b » comme suit :

4. La relation a<•b implique que le terminal « a » a une priorité inférieure au terminal « b ».

5. La relation a•>b implique que le terminal « a » a une priorité plus élevée que le terminal « b ».

6. La relation a=•b implique que le terminal « a » a une priorité inférieure au terminal « b ».

7. Un programme d'analyse de priorité d'opérateur prend une chaîne d'entrée et détermine si elle est conforme aux spécifications grammaticales. Il utilise une table d'analyse de priorité d'opérateur et une pile pour arriver à la décision.

Par exemple, si la grammaire est :

E → E+E
E → E-E
E → E*E
E → E/E
E → E^E
E → -E
E → (E)
E → id , Construct operator precedence table and accept input string " id+id*id"

Les relations de priorité entre les opérateurs sont (id) > (^) > (* /) > (+ -) > $, l'opérateur « ^ » est associatif à droite et tous les opérateurs d'alésage sont associatifs à gauche.

	+	-	*	/	^	id	(	)	$
+	•>	•>	<•	<•	<•	<•	<•	•>	•>
-	•>	•>	<•	<•	<•	<•	<•	•>	•>
*	•>	•>	•>	•>	<•	<•	<•	•>	•>
/	•>	•>	•>	•>	<•	<•	<•	•>	•>
^	•>	•>	•>	•>	<•	<•	<•	•>	•>
Id	•>	•>	•>	•>	•>	Err	Err	•>	•>
(	<•	<•	<•	<•	<•	<•	<•	=	Err
)	•>	•>	•>	•>	•>	Err	Err	•>	•>
$	<•	<•	<•	<•	<•	<•	<•	Err	Err

Tableau 5.3 : Les relations de précédence « id+id*id »

L'intention des relations de priorité est de délimiter le handle de la chaîne d'entrée donnée en marquant l'extrémité droite du handle. Action d'analyse : pour localiser le handle, les étapes suivantes sont suivies :

1. Ajoutez le symbole $ aux deux extrémités de la chaîne d'entrée donnée.
2. Parcourez la chaîne d'entrée de gauche à droite jusqu'à ce que le •> le plus à droite soit rencontré.
3. Balayez vers la gauche sur toutes les priorités égales jusqu'à ce que la première soit une poignée.
4. Tout ce qui se trouve entre <• et •> est une poignée.
5. $ sur S signifie que l'analyse est un succès.

Algorithme d'analyse de l'opérateur :

Le programme d'analyse de priorité des opérateurs Parser détermine l'action de l'analyseur en fonction de

1. a=b=$, l'analyse est réussie
2. a <• b ou a = b, l'analyseur déplace le symbole d'entrée sur la pile et avance le pointeur d'entrée vers le symbole d'entrée suivant.
3. a •> b, l'analyseur effectue l'action de réduction. L'analyseur extrait les éléments un par un de la pile jusqu'à ce que nous trouvions que l'élément supérieur actuel de la pile a une priorité inférieure à celle du terminal le plus récemment sorti.

Par exemple, la séquence d'actions entreprises par l'analyseur utilisant la pile pour la chaîne d'entrée « id * id » et l'arbre d'analyse correspondant sont les suivantes.

STACK	INPUT	OPERATIONS
$	id * id $	$ <• id, shift 'id' in to stack
$ id	*id $	id •> *, reduce 'id' using E-> id
$E	*id $	$ <• *, shift '*' in to stack
$E*	id$	* <• id , shift 'id' in to Stack
$E*id	$	id •> $, reduce 'id' using E->id
$E*E	$	*•> $, reduce '*' using E->E*E
$E	$	$=$=$, so parsing is successful

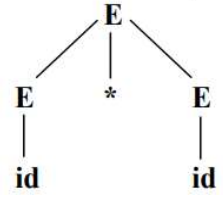


Tableau 5.4 : Séquence d'actions entreprises par l'analyseur utilisant la pile pour la chaîne d'entrée « id * id » et l'arbre d'analyse

4. Analyseurs LR

Les analyseurs LR (Left-to-Right, Rightmost derivation) sont une classe d'analyseurs syntaxiques utilisés dans le processus de compilation des langages de programmation. Ils effectuent une analyse syntaxique ascendante (ou bottom-up parsing) en commençant par les symboles terminaux (tokens) du programme source et en construisant progressivement l'arbre syntaxique en remontant vers la racine. Les analyseurs LR sont largement utilisés car ils peuvent gérer un large éventail de grammaires formelles, y compris des grammaires ambigües, et sont capables de générer un arbre syntaxique de manière efficace.

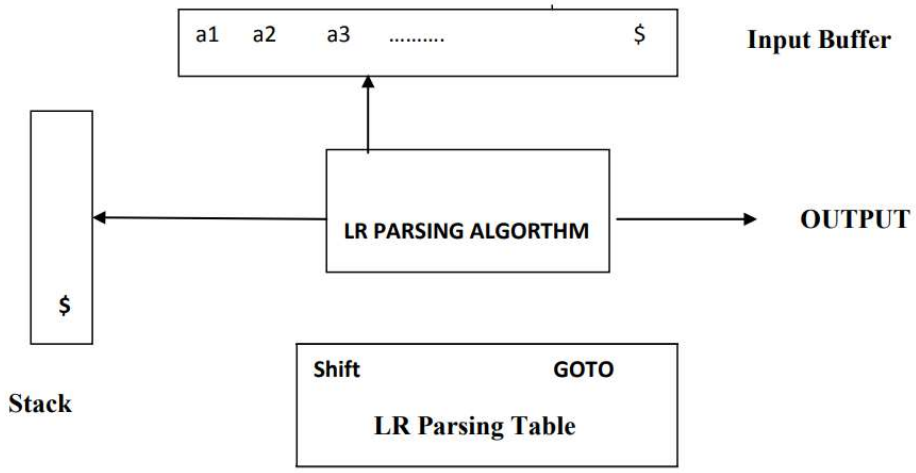


Figure 5.1: Components of LR Parsing

L'analyseur LR se compose de • Un tampon d'entrée qui contient la chaîne à analyser suivie d'un symbole $, utilisé pour indiquer la fin de l'entrée.

• Une pile contenant une séquence de symboles de grammaire avec un $ en bas de la pile, qui contient initialement l'état initial de la table d'analyse au-dessus de $.

• Une table d'analyse (M), c'est un tableau à deux dimensions M[état, terminal ou Non terminal] et il contient deux parties :

1. Partie **ACTION**

La partie ACTION de la table est un tableau à deux dimensions indexé par l'état et le symbole d'entrée, c'est-à-dire ACTION [state][input], une entrée de table d'action peut contenir l'un des quatre types de valeurs suivants. Ils sont :

1. Shift X, où X est un numéro d'État.
2. Réduisez X, où X est un numéro de production.
3. Acceptez, signifiant la fin d'une analyse réussie.
4. Entrée d'erreur.

2. Partie **GO TO**

La partie GO TO de la table est un tableau à deux dimensions indexé par état et un Non terminal, c'est-à-dire GOTO[state][NonTerminal]. Une entrée GO TO a un numéro d'état dans le tableau.

• Un algorithme d'analyse utilise l'état X actuel, le symbole d'entrée suivant « a » pour consulter l'entrée à l'action [X] [a]. il effectue l'une des quatre actions suivantes, comme indiqué ci-dessous :

1. Si l'action[X][a]=shift Y, l'analyseur exécute un décalage de Y vers le haut de la pile et avance le pointeur d'entrée.
2. Si l'action[X][a]= réduire Y (Y est le numéro de production réduit dans l'état X), si la production est Y->β, alors l'analyseur extrait 2*β symboles de la pile et appuie sur Y. sur la pile.
3. Si l'action[X][a]= accept, alors l'analyse est réussie et la chaîne d'entrée est acceptée.
4. Si l'action[X][a]= erreur, alors l'analyseur a découvert une erreur et appelle la routine d'erreur.

L'analyse est classée en

1. LR (0)
2. LR-SLR simple (1)
3. Canonique LR-CALR (1)
4. Regarder vers l'avenir - LALR (1)

4.1 LR (1) Analyse

Différentes étapes impliquées dans l'analyse LR(1) :

1. Écrivez la grammaire sans contexte pour la chaîne d'entrée donnée.
2. Vérifiez l'ambiguïté.
3. Ajouter une production augmentée.
4. Créer une collection canonique d'éléments LR(0).
5. Dessinez DFA.
6. Construisez la table d'analyse LR(0).
7. Basé sur les informations du tableau, avec l'aide de pile et l'algorithme d'analyse.

4.1.1 Augmenter la grammaire

L'Augment Grammar G`, est G avec un nouveau symbole de départ S` et une production supplémentaire S`-> S. Cela aide l'analyseur à identifier quand arrêter l'analyse et à annoncer l'acceptation de l'entrée. La chaîne d'entrée est acceptée si et seulement si l'analyseur est sur le point de réduire de S`-> S. Par exemple, considérons la grammaire ci-dessous :

E → E+T|T
T → T*F
F → (E) | id **the Augment grammar G` is Represented by**

E`→ E
E → E+T|T
T → T*F
F → (E) | id

4.1.2 Éléments LR (0)

Un élément LR (0) d'une grammaire est une production G avec un point à une certaine position sur le côté droit de la production. Un élément indique la quantité d'entrée qui a été analysée jusqu'à un point donné dans le processus d'analyse. Par exemple, si la Production est X YZ alors, les éléments LR (0) sont :

1. X-> •AB, indique que l'analyseur attend une chaîne dérivée de AB.
2. X-> A•B, indique que l'analyseur a analysé la chaîne dérivée de A et attend la chaîne de Y.
3. X-> AB•, indique que l'analyseur a analysé la chaîne dérivée de AB. Si la grammaire est X-> ε the, l'item LR (0) est X -> •, indiquant que la production est réduite à un.

4.1.3 Collection canonique d'éléments LR(0)

Il s'agit du processus de regroupement des éléments LR (0) en fonction des opérations de fermeture et d'accès.

4.1.3. a) Opération de fermeture

Si I est un Etat initial, alors la Clôture (I) est construite comme suit :

1. Dans un premier temps, ajoutez Augmenter la production à l'état et vérifiez le symbole • dans la production de droite, si le • est suivi d'un non-terminal, puis ajoutez les productions qui déclarent avec ce non-terminal dans l'état I.

2. Si une production X -> α•Aβ est dans I, alors ajoutez les productions qui commencent par X dans l'État I. La règle 2 est appliquée jusqu'à ce qu'aucune production ne soit ajoutée à l'État I (ce qui signifie que le • est suivi d'un Terminal symbole).

Exemple :

0. E`→ E		E`→• E
1. E→ E+T	LR (0) items for the Grammar is	E →•E+T
2. T→ F		T→ •F
3. T→ T*F		T→ •T*F
4. F→ (E)		F→ • (E)
5. F→ id		F→ • id

État de fermeture (I_0)

Ajoutez E ` -> • E dans l'état I_0 .Puisque, le symbole '•' dans la production de droite est suivi d'un E non terminal. Ajoutez donc les productions commençant par E dans l'état I_0. L'État devient alors :

E ` • E →

0. E→ •E+T
1. T → •F

Les 1ère et 2ème productions satisfont à la 2ème règle. Ajoutez donc les productions qui commencent par E et T dans I_0.

Remarque : une fois les productions ajoutées dans l'état, la même production ne doit pas être ajoutée une 2ème fois dans le même état. Ainsi, l'État devient.

0. E`→ •E
1. E→ •E+T
2. T→ •F
3. T→ •T*F
4. F→ •(E)
5. F→ • id

4.1.3. b) Opération GO TO

Goto(I_0, X), où I_0 est un ensemble d'éléments et X est le symbole de grammaire sur lequel nous déplaçons le symbole « • ». C'est comme trouver l'état suivant du NFA pour un état I_0 donné et le symbole d'entrée est X.

Par exemple, si la production est E -> •E+T.

Goto (I0, E) est E`-> •E, E-> E•+T

Remarque : Une fois l'opération Go to terminée, nous devons calculer l'opération de fermeture pour la production en sortie.

Goto (I0, E) est E -> E•+T, E`-> E• = Fermeture ({E` -> E•, E ->E•+T})

4.1.4 Construction de la table d'analyse LR (0)

La construction de la table d'analyse LR(0) est une étape importante dans la mise en œuvre d'un analyseur syntaxique ascendant (bottom-up) pour une grammaire contextuelle libre. Une table d'analyse LR(0) est utilisée pour guider le processus d'analyse et prendre des décisions sur le décalage (shift) et la réduction (reduce) pendant l'analyse syntaxique.

Algorithme de construction de la table d'analyse SLR (LR(0) ou SLR(1)) :

Entrée : Une grammaire augmentée G'

Sortie : La table d'analyse SLR fonctionne avec l'action et le goto pour G'

Méthode :

1. Construire C = $\{I_0, I_1, \ldots I_n\}$, la collection d'ensembles d'éléments LR(0) pour G'.
2. L'état i est construit à partir de I_i.. Les fonctions d'analyse pour l'état i sont déterminées comme suit :
   (a) Si $[A \rightarrow \alpha \cdot a\beta]$ est dans I_i et goto(I_i,a) = I_j, alors définissez l'action[i,a] sur « shift j ». Ici, un terminal incontournable.
   (b) Si $[A \rightarrow \alpha \cdot]$ est dans I_i , alors définissez l'action[i,a] sur « réduire $A \rightarrow \alpha$ » pour tout a dans FOLLOW(A).
   (c) Si [S'→S.] est dans Ii, alors définissez l'action[i,$] sur « accepter ».
   Si des actions contradictoires sont générées par les règles ci-dessus, nous disons que la grammaire n'est pas SLR(1).
3. Les transitions goto pour l'état i sont construites pour tous les non-termes
   Si goto(Ii,A) = Ij, alors goto[i,A] = j.
4. Toutes les entrées non définies par les règles (2) et (3) sont faites « erreur »
5. L'état initial de l'analyseur est celui construit à partir du [S'→.S].

Exemple sur SLR (1) Grammaire

S → E

E → E + T | T

T → T * F | F

F → id

Augmenter la grammaire et insérez le symbole « • » à la première position pour chaque règle en G.

S` → •E

E → •E + T

E → •T

T → • T * F

T → •F

F → •id

État I_0 : Ajouter la production d'augmentation à l'état I_0 et calculer la fermeture

I_0 = Closure (S` → •E)

Ajoutez toutes les productions commençant par E dans l'état I_0 car "." est suivi du non-terminal. Ainsi, l'état I_0 devient :

I0 = S` → •E

E → •E + T

E → •T

Ajoutez toutes les productions commençant par T et F dans l'état I_0 modifié car "." est suivi du non-terminal. Ainsi, l'État I_0 devient.

I0= S` → •E

E → •E + T

E → •T

T → •T * F

T → •F

F → •id

I1= Go to (I0, E) = closure (S` → E•, E → E• + T)

I2= Go to (I0, T) = closure (E → T•T, T• → * F)

I3= Go to (I0, F) = Closure (T → F•) = T → F•

I4= Go to (I0, id) = closure (F → id•) = F → id•

I5= Go to (I1, +) = Closure (E → E +•T)

Ajoutez toutes les productions commençant par T et F dans l'état I_5 car "." est suivi du non-terminal. Ainsi, l'état I_5 devient

I5 = E → E +•T

T → •T * F

T → •F

F → •id

Go to (I5, F) = Closure (T → F•) = (same as I3)

Go to (I5, id) = Closure (F → id•) = (same as I4)

I6= Go to (I2, *) = Closure (T → T * •F)

Ajoutez toutes les productions commençant par F dans l'état I_6 car "." est suivi du non-terminal. Ainsi, l'état I_6 devient

I6 = T → T * •F

F → •id

Go to (I6, id) = Closure (F → id•) = (same as I4)

I7= Go to (I5, T) = Closure (E → E + T•) = E → E + T•

I8= Go to (I6, F) = Closure (T → T * F•) = T → T * F•

Dessiner un diagramme à états finis DEF

Le DEF suivant donne les transitions d'état de l'analyseur et il est utile pour construire la table d'analyse LR.

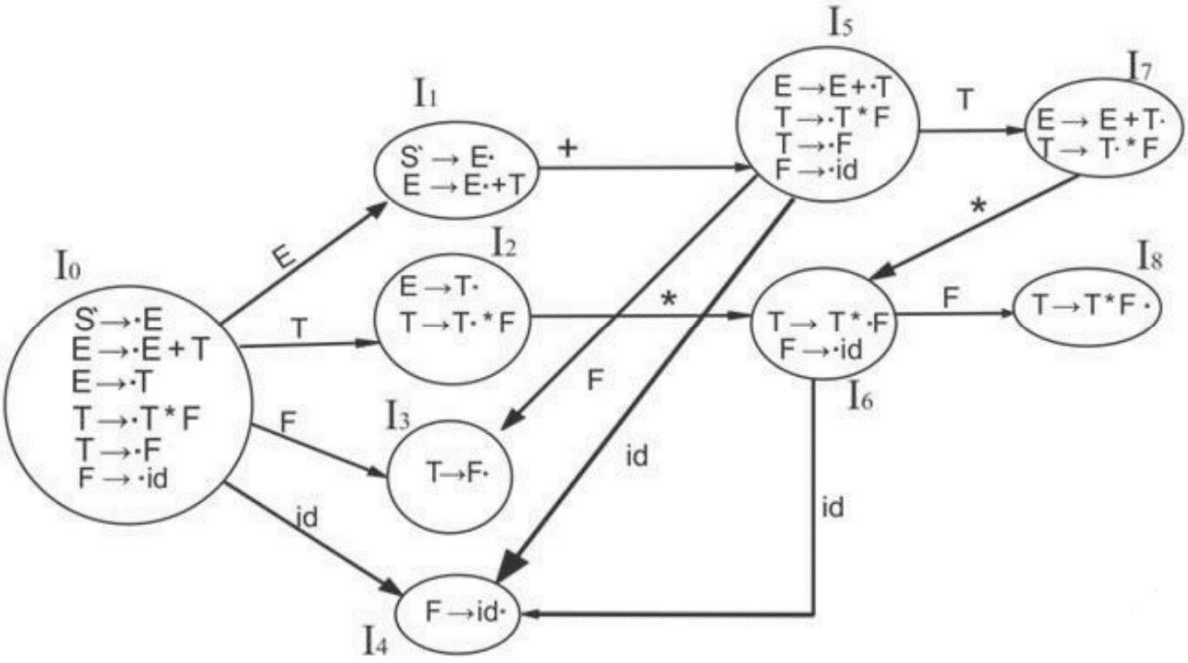


Figure 5.2 : Diagramme à états finis DEF

Tableau SLR (1):

States	Action				Go to		
	id	+	*	$	E	T	F
I_0	S_4				1	2	3
I_1		S_5		Accept			
I_2		R_2	S_6	R2			
I_3		R_4	R_4	R4			
I_4		R_5	R5	R5			
I_5	S4					7	3
I_6	S4						8
I_7		R1	S6	R1			
I_8		R3	R3	R3			

Tableau 5.5 : La table d'analyse LR.

Algorithme d'analyse LR

Entrée : Chaîne w et table d'analyse LR

Sortie : Résultat de l'analyse ascendante de w si w ∈ L(G) (arbre syntaxique) ou échec si w∉L(G).

Méthode : on commence avec S0 dans la pile, w$ dans le tampon d'entrée, le pointeur source ps est initialisé au début de w. L'analyseur exécute la boucle Répéter jusqu'à ce qu'il rencontre une action Accepter ou Erreur.

Algorithme Analyse LR;
Répéter indéfiniment
Début
Soit S l'état en sommet de pile et a le symbole pointé par ps
Si Action [S, a]= Décaler S' **Alors**
Début
Empiler a puis S' ;
Avancer ps ;
Fin ;
Sinon Si Action [S, a]= Réduire par A→β **Alors**
Début
Dépiler 2|β| symboles ;
Soit S' le nouveau sommet de pile ;
Empiler A puis Successeur[S', A] ;
Emettre en sortie A→β ;

Fin

Sinon Si Action [S, a]= Accepter **Alors** Return **Sinon** Erreur() ;

Fin

Explication :

First (E) = First (E + T) ∪ First (T)

First (T) = First (T * F) ∪ First (F)

First (F) = {id}

First (T) = {id}

First (E) = {id}

Follow (E) = First (+T) ∪ {$} = {+, $}

Follow (T) = First (*F) ∪ First (F)

= {*, +, $}

Follow (F) = {*, +, $}

1) I1 contient l'élément final qui pilote :

   S → E• et follow (S) = {$}, donc action {I1, $} = Accepter

2) I2 contient l'élément final qui pilote :

   E → T• et follow (E) = {+, $}, donc action {I2, +} = R2, action {I2, $} = R2

3) I3 contient l'élément final qui pilote :

   T → F• et follow (T) = {+, *, $}, donc action {I3, +} = R4, action {I3, *}
   = R4, action {I3, $} = R4

4) I4 contient l'élément final qui pilote :

   F → id• et follow (F) = {+, *, $}, donc action {I4, +} = R5, action {I4, *}
   = R5, action {I4, $} = R5

5) I7 contient l'élément final qui pilote :

   E → E + T• et follow (E) = {+, $}, donc action {I7, +} = R1, action {I7, $} = R1

6) I8 contient l'élément final qui pilote :

   T → T * F• et follow (T) = {+, *, $}, donc action {I8, +}= R3, action {I8, *}
   = R3, action {I8, $} = R3.

Chapitre 06 :

Traduction Dirigée par la Syntaxe

1. Introduction

La "traduction dirigée par la syntaxe" (en anglais, "Syntax-Directed Translation") est une technique utilisée dans le processus de compilation des langages de programmation pour associer des actions sémantiques au processus d'analyse syntaxique. Cette technique permet de construire une représentation interne du programme source tout en générant du code intermédiaire ou en effectuant d'autres tâches liées à la sémantique du programme.

Cette technique constitue une approche fondamentale pour exécuter diverses analyses sémantiques et faciliter les processus de traduction de code. Dans une définition orientée syntaxe, deux composants fondamentaux sont établis :

a) ***Attributs associés aux symboles grammaticaux*** : chaque symbole grammatical, englobant à la fois les terminaux et les non-terminaux, est accompagné d'un ensemble d'attributs. Ces attributs jouent un rôle crucial dans la transmission d'informations sur la structure et les propriétés du programme.

b) ***Règles sémantiques pour le calcul des attributs*** : pour chaque production de la grammaire, un ensemble correspondant de règles sémantiques est défini. Ces règles dictent la manière de calculer les valeurs des attributs liés aux symboles impliqués dans cette production.

Lorsqu'une grammaire incorpore des attributs et des règles sémantiques de cette manière, on parle de grammaire attribuée. Le résultat de l'utilisation de cette approche est un arbre d'analyse qui a été augmenté de valeurs d'attribut à chacun de ses nœuds, le transformant en un arbre d'analyse annoté. Ces valeurs d'attribut sont essentielles à l'exécution d'un large éventail de tâches, allant des analyses sémantiques à la génération de code, faisant ainsi de la traduction

dirigée par la syntaxe une technique indispensable dans la construction du compilateur et le traitement du langage de programmation.

2. Types d'attributs

Dans le contexte de la traduction dirigée par la syntaxe et de la construction d'arbres de syntaxe abstraite lors de l'analyse syntaxique d'un langage de programmation, les attributs sont des informations ou des valeurs associées aux nœuds de l'arbre syntaxique. Il existe deux types principaux d'attributs : les attributs hérités (inherited attributes) et les attributs synthétisés (synthesized attributes). Chacun de ces types d'attributs a un rôle différent dans la traduction dirigée par la syntaxe.

2.1 Attributs Hérités (Inherited Attributes)

- Les attributs hérités sont passés de nœuds parents à nœuds enfants dans l'arbre syntaxique.
- Ils sont utilisés pour propager des informations de nœuds parents aux nœuds enfants.
- Les attributs hérités sont généralement associés à des règles de production ascendantes (bottom-up) de la grammaire.
- Par exemple, dans une grammaire pour un langage de programmation, un attribut hérité peut être utilisé pour transmettre le type d'une expression depuis un nœud parent (par exemple, un opérateur arithmétique) vers ses nœuds enfants (par exemple, les opérandes).

Exemple :

Production	Semantic Rules
$D \rightarrow T$ **id** L ;	*enter*(**id**.*name*, *T.type*) *L.type* := *T.type*
$T \rightarrow$ **int**	*T.type* := INT_TYPE
$T \rightarrow$ **float**	*T.type* := FLOAT_TYPE
$L \rightarrow$, **id** L^2	enter(**id**.*name*, L.*type*) L^2.*type* := *L.type*
$L \rightarrow \varepsilon$	

Tableau 6.1 : Exemple des règles sémantique

L'arbre d'analyse annoté pour « float id, id, id »; à la grammaire ci-dessus est :

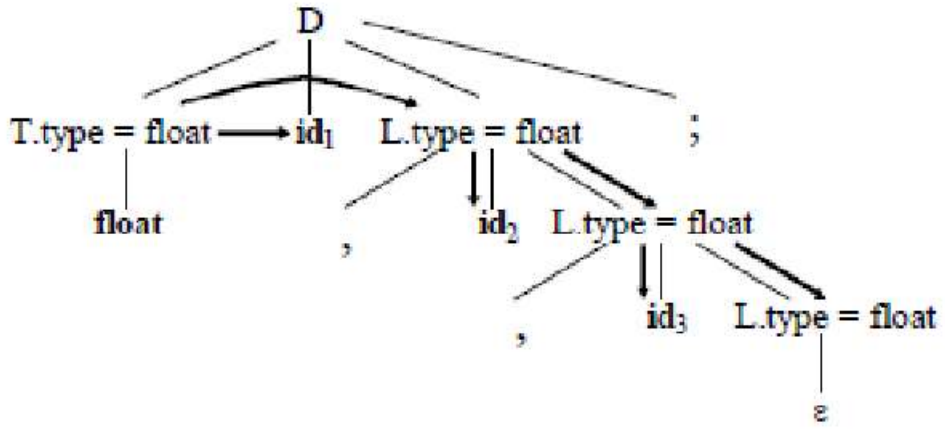


Figure 6.1 : Exemple d'arbre d'analyse annoté pour « float id, id, id »

2. 2 Attributs Synthétisés (Synthesized Attributes)

- Les attributs synthétisés sont calculés au niveau des nœuds enfants et propagés vers le nœud parent.
- Ils sont utilisés pour calculer des informations au niveau des nœuds enfants et les agréger pour obtenir une valeur ou une information au niveau du nœud parent.
- Les attributs synthétisés sont généralement associés à des règles de production descendantes (top-down) de la grammaire.
- Par exemple, dans une grammaire pour un langage de programmation, un attribut synthétisé peut être utilisé pour calculer la valeur d'une expression arithmétique en agrégeant les valeurs de ses opérandes.

Les attributs hérités et synthétisés sont essentiels pour réaliser la traduction dirigée par la syntaxe, car ils permettent de propager des informations sémantiques à travers l'arbre syntaxique pendant le processus d'analyse syntaxique. Cette approche permet de générer du code intermédiaire, de vérifier la sémantique statique du programme, de construire un arbre de syntaxe abstraite ou d'effectuer d'autres tâches liées à la sémantique du langage de programmation. La manière dont ces attributs sont définis, calculés et propagés dépend de la conception spécifique du compilateur ou de l'interprète pour le langage source.

Considérons la grammaire hors contexte suivante qui peut décrire un langage composé de multiplication et d'addition d'entiers :

Expr → Expr + Term

Expr → Term

Term → Term * Factor

Term → Factor

Factor → "(" Expr ")"

Factor → integer

La grammaire d'attribut suivante peut être utilisée pour calculer le résultat d'une expression écrite dans la grammaire qui utilise uniquement des valeurs synthétisées et est donc une grammaire S-attribuée.

Expr1 → Expr2 + Term [Expr1.value = Expr2.value + Term.value]

Expr → Term [Expr.value = Term.value]

Term1 → Term2 * Factor [Term1.value = Term2.value * Factor.value]

Term → Factor [Term.value = Factor.value]

Factor → "(" Expr ")" [Factor.value = Expr.value]

Factor → integer [Factor.value = strToInt (integer.str)]

Exemple:

Production	Semantic Rules
E→E+T	E.val = E1.val + T.val
E→T	E.val = T.val
T→T*F	T.val = T1.val + F.val
T→F	T.val = F.val
F→(E)	F.value = E.val
F→num	F.val = num.val

Un arbre d'analyse, montrant la ou les valeurs de son ou ses attributs, est appelé arbre d'analyse annoté. L'arbre d'analyse annoté pour 5+2*3 de la grammaire ci-dessus est :

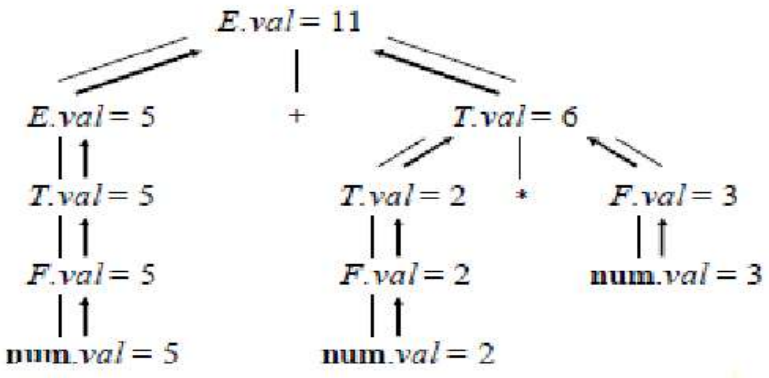


Figure 6.2 : Exemple d'arbre d'analyse annoté pour 5+2*3

3. Code intermédiaire

Le code intermédiaire est une représentation intermédiaire du code source d'un programme qui est utilisée dans le processus de compilation. Il s'agit d'une forme intermédiaire entre le code source d'origine et le code machine final. Le code intermédiaire est généré pendant la phase d'analyse syntaxique et est utilisé pour simplifier et normaliser la sémantique du programme avant de générer le code machine cible.

3.1 Structure logique d'un Front End de compilateur

Un Front End de compilateur est organisé comme dans la figure ci-dessus, où l'analyse, la vérification statique et la génération de code intermédiaire sont effectuées séquentiellement ; parfois, ils peuvent être combinés et intégrés en analyse. Tous les schémas peuvent être implémentés en créant un arbre syntaxique et en parcourant l'arbre.

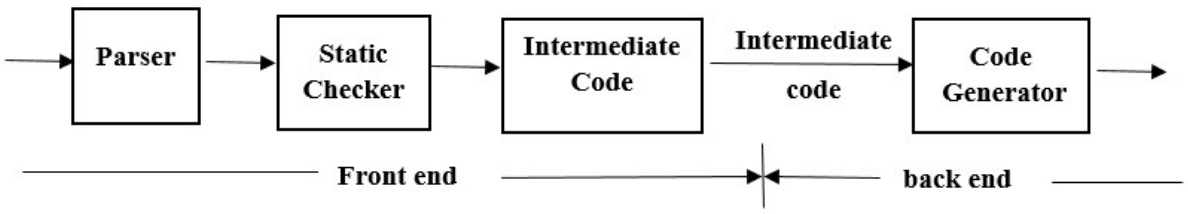


Figure 6.3 : Un Front End de compilateur

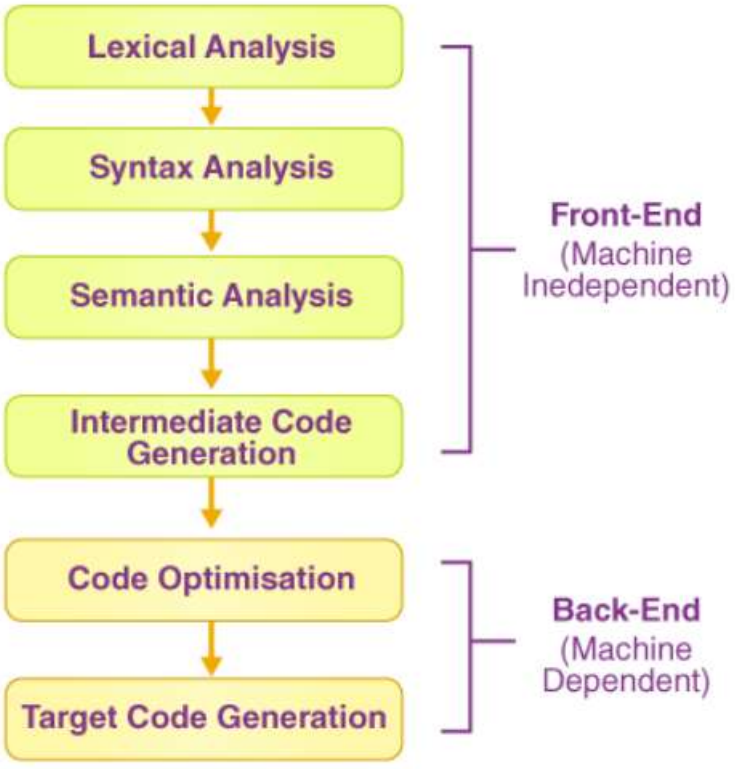


Figure 6.4 : Structure logique d'un compilateur

La vérification statique inclut la vérification de type, qui garantit que les opérateurs sont appliqués aux opérandes compatibles. Dans le processus de traduction d'un programme dans un langage source donné en code pour une machine cible donnée, un compilateur construit une séquence de représentations intermédiaires.

Les représentations de haut niveau sont proches du langage source et les représentations de bas niveau sont proches de la machine cible. Une représentation de bas niveau convient aux tâches dépendantes de la machine telles que l'allocation de registres et la sélection d'instructions.

Le processus de traduction est généralement divisé en deux étapes :

a) La génération d'un code intermédiaire avec un niveau de complexité compris entre celui du code source et celui du code objet.
b) La traduction du code intermédiaire en code objet.

Il existe trois formes de codes intermédiaires :

- Code à trois adresses (TAC).
- Notations de suffixes et de préfixes.
- Arbres de syntaxe abstraite (AST).

Ces formes de codes intermédiaires sont utilisées à différentes étapes du processus de compilation pour faciliter la génération, l'optimisation et l'analyse efficaces de code.

3.2 Code à trois adresses

Dans un code à trois adresses, il y a au plus un opérateur à droite d'une instruction ; c'est-à-dire qu'aucune expression arithmétique construite n'est autorisée. Ainsi, une expression en langage source telle que x+y*z pourrait être traduite en une séquence d'instructions à trois adresses.

$t_1 = y * z$

$t_2 = x + t1$

Où t_1 et t_2 sont des noms temporaires générés par le compilateur.

Le code à trois adresses est représenté sous forme de structure d'enregistrement avec des champs pour l'opérateur et les opérandes. Ces enregistrements peuvent être stockés sous forme de

tableau ou de liste chaînée. Les implémentations les plus courantes de trois codes d'adresse sont les **quadruples**, les **triples** et les **triples indirects**.

3.2.1 Quadruples

Les quadruplets, également appelés "quadruples" en français, sont une forme de représentation intermédiaire du code d'un programme. Ils sont utilisés dans le processus de compilation pour représenter les opérations et les expressions d'un programme de manière simplifiée et indépendante de l'architecture matérielle cible. Les quadruplets sont généralement composés de quatre parties principales :

a. **Opération (op)** : L'opération indique l'opération ou l'instruction à effectuer, telle que "addition", "soustraction", "multiplication", "division", etc.
b. **Opérande 1 (arg1) :** Il s'agit du premier opérande de l'opération, souvent désigné par une adresse ou un identifiant de variable.
c. **Opérande 2 (arg2) :** Le deuxième opérande de l'opération, également désigné par une adresse ou un identifiant de variable.
d. **Résultat (res) :** Le résultat de l'opération est stocké dans cette variable ou à cette adresse.

Les quadruplets sont utilisés pour représenter les opérations d'un programme de manière compacte et indépendante de la machine. Ils peuvent être générés lors de l'analyse syntaxique et sont souvent utilisés dans les phases ultérieures du processus de compilation, notamment dans l'optimisation et la génération de code machine.

Exemple : a = b + c b est représenté par arg1, c est représenté par arg2, + par op et a par res. Les opérateurs unaires comme "-" n'utilisent pas agr2. Les opérateurs comme param n'utilisent ni agr2 ni res. Pour les instructions conditionnelles et inconditionnelles, res est l'étiquette. Arg1, arg2 et res sont des pointeurs vers une table de symboles ou une table littérale pour les noms.

Exemple : a = -b * d + c + (-b) * d

Les trois codes d'adresse pour la déclaration ci-dessus sont les suivants :

t1 = - b

t2 = t1 * d

t3 = t2 + c

t4 = - b

t5 = t4 * d

t6 = t3 + t5

a = t6

Les quadruples pour l'exemple ci-dessus sont les suivants :

Op	Arg1	Arg2	Res
-	b		t1
*	t1	d	t2
+	t2	c	t3
-	b		t4
*	t4	d	t5
+	t3	t4	t6
=	t6		a

Tableau 6.2 : Les quadruples pour l'exemple a = -b * d + c + (-b) * d

3.2.2 Triples

Les "triples" sont une forme de représentation intermédiaire du code source d'un programme, utilisée dans le processus de compilation. Contrairement aux quadruplets, qui contiennent quatre composantes (opération, opérande 1, opérande 2, résultat), les triples en contiennent trois. Les triples sont généralement composés des éléments suivants :

a. **Opération** : L'opération à effectuer, similaire à celle des quadruplets. Cela peut être une opération arithmétique, une opération de contrôle de flux (comme un saut conditionnel), ou d'autres types d'opérations.
b. **Opérande 1 (arg1) :** Le premier opérande de l'opération. Il peut s'agir d'une variable, d'une constante ou d'une autre valeur.
c. **Opérande 2 (arg2) :** Le deuxième opérande de l'opération. Comme pour l'opérande 1, il peut s'agir d'une variable, d'une constante ou d'une autre valeur.

Les triples sont utilisés pour représenter le comportement du programme de manière plus abstraite que le code source d'origine, mais ils sont plus compacts que les quadruplets. Les triples sont souvent utilisés dans les étapes d'optimisation du processus de compilation, où le compilateur analyse et modifie la structure du programme pour améliorer les performances ou réduire la consommation de ressources.

Exemple : a = -b * d + c + (-b) * d

Les triples pour l'exemple ci-dessus sont les suivants :

Stmt no	Op	Arg1	Arg2
(0)	-	b	
(1)	*	d	(0)
(2)	+	c	(1)
(3)	-	b	
(4)	*	d	(3)
(5)	+	(2)	(4)
(6)	=	a	(5)

Tableau 6.3 : Les triples pour l'exemple a=-b * d + c + (-b) * d

Arg1 et arg2 peuvent être des pointeurs vers une table de symboles pour les variables du programme ou une table littérale pour une constante ou des pointeurs vers une structure triple pour les résultats intermédiaires.

Exemple : les triples pour l'instruction x[i] = y qui génère deux enregistrements sont les suivants

Stmt no	Op	Arg1	Arg2
(0)	[]=	x	i
(1)	=	(0)	y

Tableau 6.4 : Les triples pour l'instruction x[i] = y

Les triples pour l'instruction x = y[i] qui génère deux enregistrements sont les suivants

Stmt no	Op	Arg1	Arg2
(0)	=[]	y	i
(1)	=	x	(0)

Tableau 6.5 : Les triples pour l'instruction x = y[i]

Les triples sont des moyens alternatifs pour représenter l'arbre syntaxique ou le graphe acyclique dirigé pour les noms définis par le programme.

3.2.3 Indirect triples

Les triples indirects sont utilisés pour obtenir une indirection dans la liste des pointeurs. Autrement dit, il utilise des pointeurs vers des triples plutôt que la liste des triples eux-mêmes.

Exemple : a = -b * d + c + (-b) * d

	Stmt no
(0)	(10)
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)
(6)	(16)

Stmt no	Op	Arg1	Arg2
(10)	-	b	
(11)	*	d	(0)
(12)	+	c	(1)
(13)	-	b	
(14)	*	d	(3)
(15)	+	(2)	(4)
(16)	=	a	(5)

Tableau 6.6 : Les triples indirects pour a = -b * d + c + (-b) * d

3.3 Postfix et prefix notations

3.3.1 Traduire des expressions arithmétiques en Postfix

Un schéma pour traduire des expressions arithmétiques en code suffixe est le suivant :

E->E^1+E^2 {E.code:=E^1.code||E^2.code|| ‘’+’’}

E->E^1*E^2 {E.code:=E^1.code||E^2.code|| ‘’*’’}

E->id {E.code := id.nom}

Ainsi, l'expression infixe "3 + 4 * (2 - 1)" est traduite par l'expression postfixe "3 4 2 1 - * +".

3.3.2 Traduire l'affectation en Postfixe

L'affectation a la priorité la plus basse.

I-> id:=E {I.code:= id.nom||E.code||’’:=’’}

Ainsi, l'expression infixe " a:= b+c " est traduite en expression suffixe " a b c + := ".

3.3.3 Traduire des expressions conditionnelles en Postfixe

Toutes les instructions qui modifient le flux de contrôle telles que : while, for, repeat et switch peuvent être traduites à l'aide des constructions « if-then-else » et de l'instruction de branchement « goto ».

Exemple:

while a **do** b:=c **=>** 1 **if** (not a) **then** {**goto** 2 }**else** {b:=c **goto** 1} 2

Pour cela, nous utilisons les primitives postfixé suivantes :

e l bz :Connect to l if e is NULL

e_1 e_2 l blt : Connect to l if $e_1 < e_2$

e_1 e_2 l bgt : Connect to l if $e_1 > e_2$

e_1 e_2 l ble : Connect to l if $e_1 <= e_2$

e_1 e_2 bge : Connect to l if $e_1 >= e_2$

l br : Unconditional connection

Exemple:

if e=0 then a:=1 else b:=0 **=>** e l_1 bz b 0 := l_2 br ; l_1: a 1 := l_2:

3.4 Arbres de syntaxe abstraite (AST)

un arbre de syntaxe abstraite (AST) est une structure de données utilisée dans la compilation des langages de programmation pour représenter la structure syntaxique d'un programme de manière plus abstraite que l'arbre syntaxique concret. Un AST capture la sémantique du programme tout en éliminant les détails spécifiques à la syntaxe du langage source. Les AST sont couramment utilisés dans le processus de compilation pour faciliter la génération de code intermédiaire, l'optimisation, et la génération de code machine final.

Actions sémantiques dans l'analyse descendante : un exemple

parsing:

S → id := E
E → T E'
E' → + T E'
E' → ε
T → F T'
T' → * F T'
T' → ε
F → id
F → const
F → (E)

We insert the actions

S → id *{pushid}* := *{pushassn}* E *{buildassn}*
E → T E'
E' → + *{pushop}* T *{buildexpr}* E'
E' → ε
T → F T'
T' → * *{pushop}* F *{buildterm}* T'
T' → ε
F → id *{pushid}*
F → const *{pushconst}*
F → (E) *{pushfactor}*

Chapitre 07 :
Analyse Sémantique

1. Traduction dirigée par la syntaxe pour les instructions d'affectation

Dans la traduction dirigée par la syntaxe, l'instruction d'affectation concerne principalement les expressions. L'expression peut être de type réel, entier, tableau…

Considérez la grammaire :

S → id := E

E → E1 + E2

E → E1 * E2

E → (E1)

E → id

Le m=non-terminal E a deux attributs :

- E.place, un nom qui contiendra la valeur de E, et
- E.code, la séquence d'instructions à trois adresses évaluant E.

Une fonction gen(…) pour produire une séquence de trois instructions d'adresse. Les instructions elles-mêmes sont conservées dans une structure de données, par ex. list – Opérations SDD décrites à l'aide d'un pseudo-code. La traduction dirigée par la syntaxe pour l'instruction d'affectation est donnée ci-dessous ;

Production	**Semantic Rules**
S -> id :=E	S.code := E.code || gen(id.place ' :=' E.place)
E -> E_1+E_2	E.place := newtemp ; E.code :=E_1.code ||E_2.code|| gen(E.place ' :=' E_1.place '+' E_2.place)
E -> E_1*E_2	E.place := newtemp ; E.code :=E_1.code ||E_2.code|| gen(E.place ' :=' E_1.place '*' E_2.place)
E -> -E_1	E.place := newtemp ; E.code :=E_1.code || gen(E.place ' :=' 'uninus' E_1.place)
E -> {E_1}	E.place := E_1.place; E.code:= E_1.code
E -> id	E.place := id.place; E.code :=''

Tableau 7.1 : La traduction dirigée par la syntaxe

La séquence de mouvements pour générer les trois codes d'adresse pour une instruction d'affectation est donnée par l'exemple suivant.

Exemple : générer les trois codes d'adresse pour l'instruction d'affectation A= − B*(C+D)

Input	Stack	PLACE	Generated Code
A= − B*(C+D)			
= − B*(C+D)	id	A	
− B*(C+D)	id=	A −	
B*(C+D)	id = −	A− −	
*(C+D)	id = − id	A− − B	
*(C+D)	id = − E	A− − B	T_1 = - B
*(C+D)	id = E	A− T_1	
(C+D)	id= E *	A− T_1 −	
C+D)	id= E * (	A− T_1 − −	
+D)	id= E * (id	A− T_1 − − C	
+D)	id= E * (E	A− −T_1 − − C	
D)	id= E * (E +	A− T_1 − − C −	
)	id= E * (E + id	A− T_1 − − C − D	
)	id= E * (E + E	A− T_1 − − C − D	T_2 = C + D
)	id= E * (E	A− T_1 − − T_2	
	id= E * (E)	A− T_1 − − T_2 −	
	id= E * E	A− T_1 − T_2	T_3 = T_1 * T_2
	id= E	A− T_3	A = T_3
	S	S	

Tableau 7.2 : Génération des trois codes d'adresse pour l'instruction d'affectation A= − B*(C+D)

2. Déclaration d'affectation avec des types mixtes

Les constantes et les variables seraient de types différents, par conséquent le compilateur doit soit rejeter certaines opérations en mode mixte, soit générer des instructions de coercition (conversion de mode) appropriées.

Considérez deux modes :

RÉEL

INTEGER, avec un entier converti en réel si nécessaire.

Un champ supplémentaire en traduction pour E est E.MODE dont la valeur est REAL ou INTEGER. La règle sémantique pour E.MODE associée à la production E → E1 + E2 est :

E → E1 + E2 {if E1.MODE = INTEGER and E2.MODE = INTEGER then E.MODE = INTEGER else E.MODE = REAL }

Si nécessaire les trois codes d'adresse

A=**inttoreal B** est généré,

Dont l'effet est de convertir l'entier B en un réel d'égale valeur appelé A.

Exemple : Considérez l'instruction d'affectation

X=Y+I*J

Supposons que X et Y soient RÉELS et que I et J aient le mode INTEGER.

Code à trois adresses :

T1=I int * J

T2= inttoreal T1

T3=Y real +T2

X=T3

3. Traduction dirigée par la syntaxe (SDT) pour l'expression booléenne

Les expressions booléennes ont deux objectifs principaux.

1. Ils sont utilisés pour calculer les valeurs logiques.

2. Ils sont également utilisés comme expressions conditionnelles qui modifient le flux de contrôle, comme if-then-else ou while-do.

Les expressions booléennes sont composées d'opérateurs booléens (and ,or, and not) appliqués aux éléments qui sont des variables booléennes ou des expressions relationnelles. Les expressions relationnelles sont de la forme E1 relop E2 , où E1 et E2 sont des expressions arithmétiques.

Considérez la grammaire

E → E or E

E → E and E

E → not E

E → (E)

E → id relop id

E → TRUE E → id

E → FALSE

Le relop est l'un des <,≤,=, ≠, > ,or ≥.. Nous supposons que **or** et **and** restent associatifs. **or** a priorité la plus basse, alors **and**, alors **not.**

4. Méthodes de traduction des expressions booléennes

Il existe deux méthodes pour représenter la valeur d'un booléen.

1. Pour encoder numériquement vrai ou faux
    - 1 est utilisé pour désigner vrai.
    - 0 est utilisé pour indiquer faux.
2. Évaluer une expression booléenne de manière analogue à une expression arithmétique.
    - Flux de contrôle – représentant la valeur d'une expression booléenne par une position atteinte dans un programme.
    - Utilisé dans le flux d'instructions de contrôle telles que les instructions if-then-else ou while-do.

- *Représentation numérique des expressions booléennes :*

Considérons d'abord l'implémentation d'expressions booléennes utilisant 1 pour désigner VRAI et 0 pour désigner FAUX. Les expressions seront évaluées de gauche à droite.

Exemple 1 :

A or B and C

Séquence de trois adresses :

T1 = B and C

T2 = A or T1

Exemple 2 :

Une expression relationnelle A< B est équivalente à l'instruction conditionnelle.

if A < B then 1 else 0

Séquence de trois adresses :

(1) if A < B goto (4)

(2) T=0

(3) goto (5)

(4) T=1

(5) ---

- *Règles sémantiques pour l'expression booléenne :*

Les règles sémantiques pour l'expression booléenne sont données ci-dessous.

$E \rightarrow E_1$ or E_2	{E.place = newtemp(); gen (E.place = E_1.place **or** E_2.place)}
$E \rightarrow E_1 + E_2$	{E.place = newtemp(); gen (E.place = E_1.place **and** E_2.place)}
$E \rightarrow$ NOT E_1	{E.place = newtemp(); gen (E.place = **not** E_1.place)}
$E \rightarrow (E_1)$	{E.place = E_1.place}
$E \rightarrow id_1$ relop id_2	{ E.place = newtemp(); gen (if id_1.place **relop.op** id_2.place goto nextquad + 3); gen(E.place=0) gen(goto nextquad + 2) gen(E.place=1) }
E → TRUE	{E.place := newtemp(); gen(E.place=1)}
E → FALSE	{E.place := newtemp(); gen(E.place=0)}

Tableau 7.3 : Les règles sémantiques pour l'expression booléenne.

Exemple: Code for a < b or c < d and e <f

100: if a < b goto 103

101: t1 = 0

102: goto 104

103: t1 = 1

104: if c < d goto 107

105: t2 =0

106: goto 108

107: t2 = 1

108: if e < f goto 111

109: t3 =0

110: goto 112

111: t3 = 1

112: t4 = t2 and t3

113: t5 = t1 or t4

5. Génération de table de symboles

La table de symboles est une structure de données importante créée et maintenue par les compilateurs afin de stocker des informations sur l'occurrence de diverses entités telles que les noms de variables, les noms de fonctions, les objets, les classes, les interfaces, etc. La table des symboles est utilisée à la fois par les parties d'analyse et de synthèse d'un compilateur. Il se construit en phases d'analyse lexicale et syntaxique.

Les informations sont collectées par les phases d'analyse du compilateur et sont utilisées par les phases de synthèse du compilateur pour générer du code. Il est utilisé par le compilateur pour atteindre l'efficacité du temps de compilation. Il est utilisé par différentes phases du compilateur comme suit :

- **Analyse lexicale** : crée de nouvelles entrées de tableau dans le tableau, par exemple comme des entrées sur le jeton.
- **Analyse syntaxique** : ajoute des informations concernant le type d'attribut, la portée, la dimension, la ligne de référence, l'utilisation, etc. dans le tableau.
- **Analyse sémantique** : utilise les informations disponibles dans le tableau pour vérifier la sémantique, c'est-à-dire pour vérifier que les expressions et les affectations sont sémantiquement correctes (vérification de type) et la met à jour en conséquence.
- **Génération de code intermédiaire** : fait référence à la table des symboles pour savoir combien et quel type de temps d'exécution est alloué et la table aide à ajouter des informations sur les variables temporaires.
- **Optimisation du code** : utilise les informations présentes dans la table des symboles pour les applications dépendantes de la machine. optimisation.
- **Génération de code cible** : génère du code en utilisant les informations d'adresse de l'identifiant présent dans le tableau.

Une table de symboles est simplement une table qui peut être linéaire ou une table de hachage. Il conserve une entrée pour chaque nom au format suivant :

<Symbol Name, Type, Attribute>

Par exemple, si une table de symboles doit stocker des informations sur la déclaration de variable suivante :

static int interest;

Ensuite, il doit stocker l'entrée telle que :

<interest, int, static>

La clause d'attribut contient les entrées liées au nom. Éléments stockés dans la table des symboles :

- Noms de variables et constantes
- Noms des procédures et des fonctions
- Constantes littérales et chaînes
- Temporaires générés par le compilateur
- Étiquettes dans les langues sources

Informations utilisées par le compilateur à partir de la table des symboles :

- Type et nom des données
- Déclarer les démarches
- Décalage en stockage
- Si structure ou enregistrement, alors, pointeur vers la table de structure.
- Pour les paramètres, que ce soit un paramètre passant par valeur ou par référence
- Nombre et type d'arguments passés à la fonction
- Adresse de base

Opérations de la table de symboles – Les opérations de base définies sur une table de symboles comprennent :

Operations	**Fonctions**
Allocate	Attribuer une nouvelle table de symboles vide
Free	Pour supprimer toutes les entrées et libérer le stockage de la table des symboles
Lookup	Pour rechercher un nom et renvoyer un pointeur vers son entrée
Insert	Pour insérer un nom dans une table de symboles et renvoyer un pointeur vers son entrée
Set_Attribute	Associer un attribut à une entrée donnée
Get_Attribute	Pour récupérer un attribut associé à une entrée donnée

Tableau 7.4 : Opérations de la table de symboles

Chapitre 08:
Environnement d'Exécution

1. Introduction

Un environnement d'exécution, souvent appelé runtime en anglais, est un environnement logiciel dans lequel un programme informatique s'exécute. Il fournit les ressources et les services nécessaires à l'exécution d'un programme, y compris l'accès au matériel, la gestion de la mémoire, la gestion des fichiers, les bibliothèques système et d'autres fonctionnalités essentielles.

2. Arbres d'activation

Un programme est une séquence d'instructions combinées en un certain nombre de procédures. Les instructions d'une procédure sont exécutées séquentiellement. Une procédure a un délimiteur de début et un délimiteur de fin et tout ce qu'elle contient est appelé le corps de la procédure. L'identifiant de la procédure et la séquence d'instructions finies qu'il contient constituent le corps de la procédure. L'exécution d'une procédure s'appelle son activation. Un enregistrement d'activation contient toutes les informations nécessaires pour appeler une procédure. Un enregistrement d'activation peut contenir les unités suivantes *dependinguponthesourcelanguageused*.

Temporaries	Stocke les valeurs temporaires et intermédiaires d'une expression.
Données locales	Stocke les données locales de la procédure appelée
État de la machine	Stocke l'état de la machine, tel que les registres, le compteur de programme, etc., avant l'appel de la procédure.
Control Link	Stocke l'adresse de l'enregistrement d'activation de la procédure appelante.
Access Link	Stocke les informations sur les données qui se trouvent en dehors de la portée locale.
Paramètres réels	Stocke les paramètres réels, c'est-à-dire les paramètres utilisés pour envoyer une entrée à la procédure appelée.
Valeur de retour	Stocke les valeurs de retour

Chaque fois qu'une procédure est exécutée, son enregistrement d'activation est stocké sur la pile, également appelée pile de contrôle. Lorsqu'une procédure appelle une autre procédure, l'exécution de l'appelant est suspendue jusqu'à ce que la procédure appelée termine son exécution. A ce moment, l'enregistrement d'activation de la procédure appelée est stocké sur la pile. Nous supposons que le contrôle du programme s'effectue de manière séquentielle et lorsqu'une procédure est appelée, son contrôle est transféré à la procédure appelée. Lorsqu'une procédure appelée est exécutée, elle renvoie le contrôle à l'appelant. Ce type de flux de contrôle permet de représenter plus facilement une série d'activations sous la forme d'un arbre, appelé arbre d'activation.

3. Allocation de stockage

L'environnement d'exécution gère les besoins en mémoire d'exécution pour les entités suivantes :

• ***Code :*** Il s'agit de la partie texte d'un programme qui ne change pas au moment de l'exécution. Ses besoins en mémoire sont connus au moment de la compilation.

• ***Procédures :*** Leur partie texte est statique mais elles sont appelées de manière aléatoire. C'est pourquoi le stockage en pile est utilisé pour gérer les appels et les activations de procédures.

• ***Variables :*** Les variables sont connues au moment de l'exécution uniquement, sauf si elles sont globales ou constantes.

Le schéma d'allocation de mémoire de tas est utilisé pour gérer l'allocation et la désallocation de mémoire pour les variables au moment de l'exécution.

3.1 Allocation statique

Dans ce schéma d'allocation, les données de compilation sont liées à un emplacement fixe dans la mémoire et ne changent pas lors de l'exécution du programme. Comme les besoins en mémoire et les emplacements de stockage sont connus à l'avance, le package de support d'exécution pour l'allocation et la désallocation de mémoire n'est pas requis.

3.2 Allocation de pile

Les appels de procédures et leurs activations sont gérés au moyen d'une allocation de mémoire de pile. Il fonctionne selon la méthode LIFO dernier entré, premier sorti et cette stratégie d'allocation est très utile pour les appels de procédure récursifs.

3.3 Allocation de tas

Les variables locales à une procédure sont allouées et désallouées uniquement au moment de l'exécution. L'allocation de tas est utilisée pour allouer dynamiquement de la mémoire aux variables et la récupérer lorsque les variables ne sont plus nécessaires. À l'exception de la zone mémoire allouée statiquement, la mémoire de pile et la mémoire de tas peuvent augmenter et diminuer de manière dynamique et inattendue. Par conséquent, ils ne peuvent pas disposer d'une quantité fixe de mémoire dans le système.

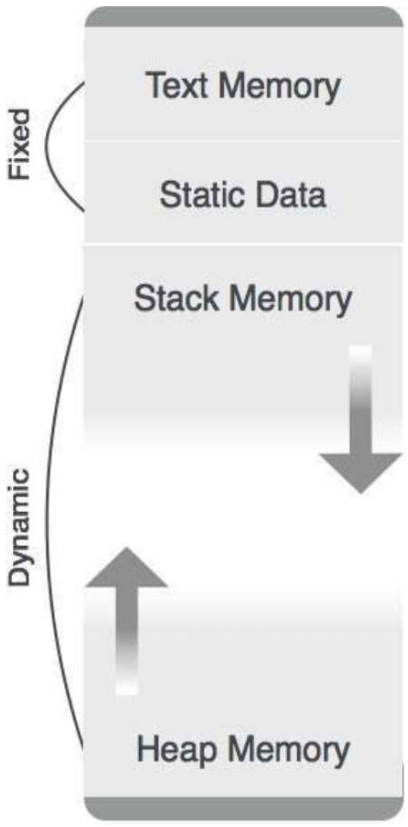


Figure 8.1 : Allocation de stockage

Comme le montre l'image ci-dessus, la partie texte du code se voit attribuer une quantité fixe de mémoire. La mémoire pile et la mémoire tas sont disposées aux extrémités de la mémoire totale allouée au programme. Les deux rétrécissent et grandissent l'un contre l'autre.

4. Passage de paramètres

Le moyen de communication entre les procédures est appelé passage de paramètres. Les valeurs des variables d'une procédure appelante sont transférées à la procédure appelée par un mécanisme. Avant d'aller de l'avant, passez d'abord en revue quelques terminologies de base relatives aux valeurs d'un programme.

• ***valeur-r*** : la valeur d'une expression est appelée sa ***valeur-r***. La valeur contenue dans une seule variable devient également une valeur r si elle apparaît à droite de l'opérateur d'affectation. Les valeurs r peuvent toujours être attribuées à une autre variable.

• ***valeur-l*** : l'emplacement de l'adresse mémoire où une expression est stockée est appelé valeur l de cette expression. Il apparaît toujours à gauche d'un opérateur d'affectation.

Par exemple :

```
jour = 1 ;
semaine = jour * 7 ;
mois = 1 ;
année = mois * 12 ;
```

À partir de cet exemple, nous comprenons que les valeurs constantes telles que 1, 7, 12 et les variables telles que le jour, la semaine, le mois et l'année ont toutes des valeurs r. Seules les variables ont des valeurs l car elles représentent également l'emplacement mémoire qui leur est attribué.

Par exemple:

```
7 = x + y;
```

est une erreur de valeur-l, car la constante 7 ne représente aucun emplacement mémoire

.• ***Paramètres formels***

Les variables qui prennent les informations transmises par la procédure appelante sont appelées paramètres formels. Ces variables sont déclarées dans la définition de la fonction appelée.

• ***Paramètres réels***

Les variables dont les valeurs ou les adresses sont transmises à la procédure appelée sont appelées paramètres réels. Ces variables sont spécifiées dans l'appel de fonction sous forme d'arguments.

Exemple:

```
fun_one()
   {
    int actual_parameter = 10;
    call fun_two(int actual_parameter);
    }
 fun_two(int formal_parameter)
{
 print formal_parameter;
 }
```

Les paramètres formels contiennent les informations du paramètre réel, en fonction de la technique de transmission des paramètres utilisée. Il peut s'agir d'une valeur ou d'une adresse.

4.1 Passer par valeur

Dans le mécanisme de transmission par valeur, la procédure appelante transmet la valeur r des paramètres réels et le compilateur place cela dans l'enregistrement d'activation de la procédure appelée. Les paramètres formels contiennent alors les valeurs transmises par la procédure appelante. Si les valeurs détenues par les paramètres formels sont modifiées, cela ne devrait avoir aucun impact sur les paramètres réels.

Exemple:

```
callByValue(int y)
{
    y = y + 1;
    print(y);
}

main()
{
    int x = 42;
    print(x);
    callByValue(x);
    print(x);
}
```

output:
x = 42
y = 43
x = 42

x's value does *not* change when y's value is changed

4.2 Passer par référence

Dans le mécanisme de passage par référence, la valeur l du paramètre réel est copiée dans l'enregistrement d'activation de la procédure appelée. De cette façon, la procédure appelée a désormais l'adresse mémoire du paramètre réel et le paramètre formel fait référence au même emplacement mémoire. Par conséquent, si la valeur indiquée par le paramètre formel est

modifiée, l'impact doit être vu sur le paramètre réel car ils doivent également pointer vers la même valeur.

Exemple:

```
fakeCallByRef(int *y)
{
     *y  = *y + 1;
     print(*y);
}

main()
{
     int x = 42;
     print(x);
     fakeCallByRef(&x);
     print(x);
}
```

must explicitly pass the address of a local variable

4.3 Passage par copie-restauration

Ce mécanisme de transmission de paramètres fonctionne de manière similaire au « passage par référence », sauf que les modifications des paramètres réels sont apportées à la fin de la procédure appelée. Lors de l'appel de la fonction, les valeurs des paramètres réels sont copiées dans l'enregistrement d'activation de la procédure appelée. Les paramètres formels, s'ils sont manipulés, n'ont aucun effet en temps réel sur les paramètres réels car les valeurs sont transmises, mais lorsque la procédure appelée se termine, les valeurs l des paramètres formels sont copiées dans les valeurs l des paramètres réels.

Exemple:

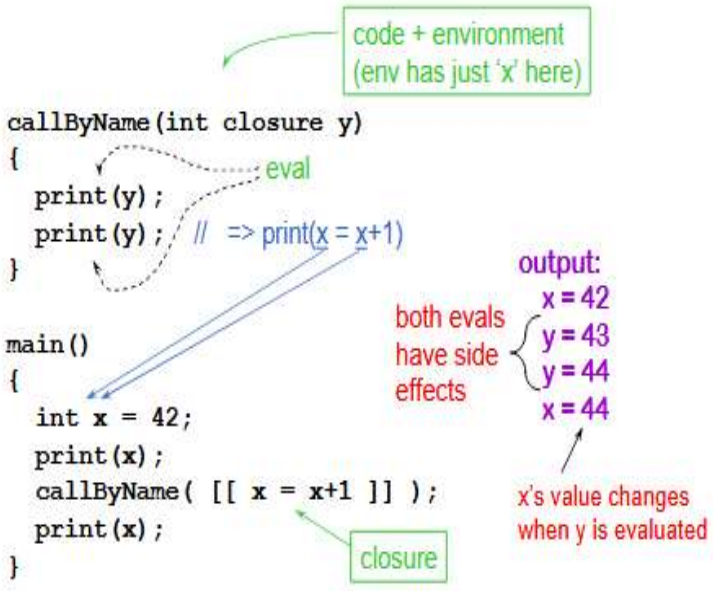
```
int y;
calling_procedure()
{
y = 10;
copy_restore(y); //l-value of y is passed
printf y; //prints 99
}
copy_restore(int x)
{
x = 99; // y still has value 10 (unaffected)
y = 0; // y is now 0}
```

Lorsque cette fonction se termine, la valeur-l du paramètre formel x est copiée dans le paramètre réel y. Même si la valeur de y est modifiée avant la fin de la procédure, la valeur-l de x est copiée dans la valeur l de y, ce qui la fait se comporter comme un appel par référence.

4.4 Passer par nom

Des langages comme Algol fournissent un nouveau type de mécanisme de transmission de paramètres qui fonctionne comme un préprocesseur en langage C. Dans le mécanisme de transmission par nom, le nom de la procédure appelée est remplacé par son corps réel. Le passage par nom remplace textuellement les expressions d'argument dans un appel de procédure par les paramètres correspondants dans le corps de la procédure afin qu'elle puisse désormais fonctionner sur des paramètres réels, un peu comme le passage par référence.

Exemple:



Chapitre 09 :
Génération et Optimisation du code

1. Introduction

Le générateur de code est la phase finale de compilateur, il a pour donnée une représentation intermédiaire du programme source et produit comme résultat un programme cible équivalent, qui doit être correct et de bonne qualité, de plus doit s'exécuter le plus rapidement possible.

2. Problèmes posés par la conception d'un générateur de code

1- Donnée du générateur de code : la donnée du générateur de code est une représentation intermédiaire du programme source couplée avec la table des symboles

2- Programmes cible : il peut prendre toute une variété de forme :

a) **langage machine absolu** : présente l'avantage de pouvoir être chargé dans un emplacement donné de la mémoire et immédiatement exécuté.

b) **langage machine translatable** : permet la compilation séparée des sous programmes.

c) **langage d'assemblage :** la production de langage d'assemblage rend le processus de production de code plus simple, puisque la production de code assembleur ne répète pas exactement la tache de l'assembleur, ce choix est une alternative raisonnable.

3- Gestion de la mémoire : A partir de la table des symboles on peut associer à un nom une adresse relative dans la zone des données.si on doit produire du code machine, on doit convertir les étiquettes des instructions à trois adresses en adresses d'instructions

4- Sélection des instructions : la sélection des instruction est simple, pour chaque type d'instruction à trois adresses, on peut élaborer un squelette de code qui fournit le profil du code machine à produire pour cette construction, malheureusement, la façon de produire du code instruction par instruction conduit souvent à du code de mauvaise qualité, par exemple, la séquence d'instruction :

a := b +c d := a + e serait traduite par :

```
MOV b, R0
ADD c, R0
MOV R0,a
MOV a, R0
ADD e, R0
MOV R0,d
```

Dans cette séquence, la quatrième instruction est redondante, ainsi que la troisième, si a n'est pas utilisé ensuite. La qualité du code produit est déterminée par sa vitesse d'exécution et sa taille. Une machine cible possédant un jeu d'instructions riche peut fournir plusieurs façons d'implanter une opération donnée, par exemple si la machine cible possède une instruction d'incrémentation (INC), l'instruction a := a+1 peut être implantée plus efficacement par la simple instruction INC a, plutôt que par :

```
MOV a,R0
ADD #1,R0
MOV R0, a
```

5- Allocation des registres : les instructions ayant des registres comme opérandes sont généralement plus courtes et s'exécutent plus rapidement que les instructions allant chercher leurs opérandes en mémoire, l'utilisation des registres est souvent divisée en deux sous problèmes :

a) pendant l'allocation des registres, on choisit l'ensemble des variables qui vont résider dans des registres en un point donné du programme.
b) pendant la phase d'assignation des registres, on choisit le registre dans lequel une variable donnée va résider.

6- Choix de l'ordre d'évaluation : l'efficacité du code cible dépend de l'ordre dans lequel on effectue les calculs, puisque certain ordre de calcul nécessite moins de registres que d'autres.

3. Un générateur de code simple

Pour des raisons de simplicité, nous supposons qu'à chaque opération d'une instruction de code intermédiaire correspond une opération de la machine cible. Nous supposons aussi qu'on peut conserver les résultats de calcul dans des registres aussi longtemps que possible, c.à.d qu'on range ces résultats en mémoire uniquement

a) Si on a besoin du registre qui les contient pour effectuer un autre calcul.

b) Juste avant un appel de procédure, une instruction de branchement, ou une instruction étiquetée. la condition (b) implique que les contenus de tous les registres doivent être sauvegardés à la fin d'un bloc de base, il est nécessaire d'effectuer ces sauvegardes car, en quittant un bloc.

On peut produire un code d'assez bonne qualité pour l'instruction simple ADD RJ, RI de cout un , dont le résultat a se trouve dans le registre RI, on peut produire cette instruction à condition que Ri contienne b, que Rj contienne c et que b soit pas actif après cette instruction. Si Ri contient b mais que c se trouve dans un emplacement mémoire, on peut produire la séquence : ADD c , Ri cout=2 ou MOV c,Rj ADD Rj, Ri cout=3. B ne soit pas actif La seconde séquence devient plus intéressante si la valeur de c est utilisée ensuite. En résumé on constate que le processus de production de code doit examiner un très grand nombre de cas et que la solution à choisir dépend du contexte dans lequel l'instruction à trois adresses a été examinée.

4. Descripteurs de registres et d'adresses

L'algorithme de production de code utilise des descripteurs pour garder trace des noms dans les registres ou les adresses : 1 : un descripteur de registre garde la trace du contenu courant des registres. On consulte le descripteur chaque fois qu'on a besoin d'un nouveau registre. 2 : un descripteur d'adresses garde la trace du contenu d'un emplacement où l'on peut trouver la valeur courante d'un nom à l'exécution. Cet emplacement peut être un registre, un emplacement dans une pile, une adresse mémoire ou un ensemble d'emplacements parmi ceux qui précédent.

5. Un algorithme de production de code

Pour chaque instruction x := y op z d'un bloc de base, on effectue les actions suivantes :

1 : Invoque la fonction *DonnerRegistre* pour déterminer l'emplacement E dans lequel le résultat du calcul y op z sera rangé. Le plus souvent E sera un registre, mais il peut être aussi un emplacement mémoire.

2 : Consulter le descripteur d'adresses de y pour déterminer y', l'emplacement courant de y. choisir de préférence un registre pour y' si la valeur de y réside à la fois en mémoire et dans un registre. si la valeur de y n'est pas déjà dans E, produire l'instruction MOV y', E .

3 : Produire l'instruction OP z', E où z' est un des emplacements courantes de z. Modifier le descripteur d'adresses de x pour indiquer que x est dans l'emplacement E. Si E est un registre modifier son descripteur pour indiquer qu'il contient la valeur de x.

4 :Si les valeurs de y et/ou z n'ont pas d'utilisation ultérieure, ne sont pas actives à la sortie du bloc, et sont dans des registres , modifier le descripteur de registres pour indiquer qu'après l'exécution de x := y op z, ces registres ne contiennent plus y et /ou z respectivement.

Un cas particulier important : x :=y . si y est dans un registre il suffit de modifier les descripteurs de registres et d'adresses pour mémoriser le fait qu'on trouve la valeur de x uniquement dans le registre contenant la valeur de y. si y n'a pas d'utilisation ultérieure , et n'est pas actif à la sortie du bloc, le registre ne contient plus la valeur de y. Si y réside uniquement en mémoire, on peut en principe enregistrer le fait que la valeur de x se trouve dans l'emplacement de y, mais cette option complique l'algorithme, puisque on ne peut pas changer la valeur de y sans sauvegarder la valeur de x.

On peut aussi produire une instruction MOV y, x , ce qui est préférable dans le cas où la valeur de x n'a pas d'utilisation ultérieure dans le bloc. Quand toutes les instructions à trois adresses d'un bloc de base ont été traitées, on range par des instructions MOV les noms qui ne sont pas dans des emplacements mémoire et qui sont actifs à la sortie du bloc. Pour ce faire, on utilise le descripteur de registre pour déterminer quels noms résident dans des registres, le descripteur d'adresses pour déterminer si l'un ou l'autre de ces noms n'est pas déjà dans un emplacement mémoire, et les informations sur les variables actives pour connaitre l'endroit où le nom doit être rangé.

5.1 La fonction DonnerRegistre

La fonction DonnerRegistre retourne un emplacement E pour recevoir la valeur de x dans l'affectation x :=y op z :

1. Si le nom y est dans un registre qui ne contient pas la valeur d'autres noms, si y n'est pas référencé et s'il n'a aucune utilisation après l'exécution de x :=y op z, la fonction DR retourne pour L le registre de y. Modifier le descripteur d'adresse de y pour indiquer que y ne réside plus dans L, le même traitement est appliqué à l'opérande z.

2. Si (1) échoue, retourner un registre libre s'il en existe un.

3. Si (2) échoue, si x a une utilisation ultérieure dans le bloc ou si op est un opérateur qui requiert un registre (tel que l'indexation), chercher un registre occupé R. Ranger le contenu de ce registre dans un emplacement mémoire M(par MOV R ,M) s'il n y est pas déjà, modifier le descripteur d'adresses pour M, et retourner R. Si R contient la valeur de plusieurs variables, une instruction MOV doit être produite pour chaque variable qui nécessite d'être sauvegardée. Un choix convenable pour le registre occupé peut être celui dont la donnée est référencée le plus loin dans le futur, ou l'un de ceux dont le contenu est aussi en mémoire.

4. Si x n'est pas utilisé plus loin dans le bloc, ou si aucun registre occupé ne convient, choisir pour E l'emplacement mémoire de x.

Exemple : d := (a –b) + (a- c) + (a – c) peut être traduite en séquence de code à trois adresses comme suit : t := a-b , u := a-c v := t + u d := t +v Où d est active à la fin du bloc , en appliquant l'algorithme de production de code : Le fait que a , b et c sont toujours présents dans la mémoire ne figure pas dans le descripteur d'adresses, nous supposons aussi que t , u et v étant des temporaires, ils ne sont pas présents dans la mémoire

Instruction	Code produit	Descripteur de registre	Descripteur d'adresses
		Registres vides	
T := a –b	MOV a, R0 SUB b , R0	R0 contient t	T est dans R0
U:= a-c	MOV a, R1 SUB c , R1	R0 contient t R1 contient u	T est dans R0 u est dans R1
V:= t + u	ADD R1 ,R0	R0 contient V R1 contient u	v est dans R0 u est dans R1
D := v + u	ADD R1 , R0 MOV R0 , d	R0 contient d	D est dans R0 d est dans R0 et dans la mémoire

Tableau 9.1 : Exemple d'utilisation de la fonction DonnerRegistre

5.2 Production de code pour d'autres types d'instructions

Les opérations d'indexation et les opérations portant sur les pointeurs sont traitées de la même manière que les opérations binaires, le tableau suivant présente la séquence de code produite pour les instructions d'affectation de la forme a := b[i] et a[i] := b.

instruction	I dans RI		I dans MI	
	Code	Coût	Code	Coût
a := b[i]	MOV b(Ri), R	2	MOV Mi, R MOV b(R), R	4
a[i] :=b	MOV b, a(Ri)	3	MOV Mi, R MOV b, a(R)	5

Tableau 9.2 : Production de code pour a := b[i] et a[i] := b.

5.3 Instruction conditionnelles

Il existe deux implantations possibles des instructions conditionnelles. L'une d'elles consiste à effectuer le branchement en fonction du contenu d'un registre spécifique qui dénote six conditions : négatif, nul, positif, non négatif, non positif et non nul. Sur une telle machine l'instruction à trois adresses : si x <y aller à z peut être implantée en soustrayant y de x dans le registre R, puis en se branchant vers z si la valeur de R est négative. La seconde approche utilisée sur la plupart des machines, utilise un ensemble de codes conditions pour indiquer si la dernière quantité calculée et chargée dans un registre est négative, nulle ou positive, par exemple CMP x,y positionne le code-condition à une valeur positive si x>y. une instruction de branchement conditionnel effectue le branchement si la condition <,=,>, <= ;=> ou ≠ est vérifiée. Nous utilisons l'instruction machine CJ <= z qui signifie « se brancher à z si le code condition est négatif ou nul » par exemple ; si x<y aller à z pourrait être implanté par : CMP x, y CJ < z

6. Optimisation du code

Apres la génération de la forme intermédiaire du programme source une dernière phase reste à développer, c'est la génération du code objet, certaine compilateur intègre à ce niveau un autre processus qui consiste à optimiser du code intermédiaire.

La phase de l'optimisation est facultative car elle n'intervient en aucune façon dans le processus de traduction. L'optimisation du code objet revient à déterminer le plus petit code du point de

vue taille qui s'exécute le plus rapidement possible. La complexité de ce problème est la cause primordiale de l'inexistence d'une solution à ce problème (juste une amélioration en terme d'espace mémoire et temps d'exécution).

Les opérations d'optimisation peuvent être entreprises à tout moment, le long du processus de compilation et même avant celui-ci, c'est à dire pendant la tache de programmation,

Pour ce chapitre, le processus d'optimisation est effectué après la génération du code intermédiaire. Il existe plusieurs différentes opérations d'optimisation, parmi ces opérations, nous citons :

- *Emploi optimal des registres*: la bonne gestion des registres.
- *Optimisation des boucles* : réduire le temps d'exécution des boucles influencera énormément sur le temps total d'exécution du programme.
- *Optimisation locale* : remplacer le code.

6.1 Exemple d'utilisation d'instructions particulaire

a) *Cas de division et multiplication par deux :*
b) *Cas des itérations avec un pas positif :*
c) *Cas des optimisations des expressions logique*
d) *Optimisation d'indexation*
e) *Évaluation pendant la compilation*
f) *Optimisation des itérations*

6.2 Technique automatique d'optimisation des boucles

Avant d'entamer ce type d'opération, il est nécessaire de savoir détecter les boucles d'un code intermédiaire pour rendre plus complet le processus d'optimisation, la procédure de détection des boucles d'un code nécessite la connaissance de plusieurs notions à savoir les blocs fondamentaux. Ces notions vont nous permettra de construire un graphe appelé graphe de flot de contrôle :

6.2.1 Trouver les leaders

Un **leader** est une instruction particulière du code obéissant à la définition suivante :

- La première instruction du code est un leader.
- Toute instruction qui suit dans l'ordre d'apparition des instructions une instruction de branchement conditionnel est un leader.
- Toute instruction dont le numéro du quadruplet figurant comme la cible d'une instruction de branchement conditionnel ou inconditionnel est un leader.

Exemple : considérons le code intermédiaire du programme suivant :

P :=0 ; i := 1 ; While i<= n begin p := p + a[i] *b[i] ; i: = i + 1 ; end ;

1: p:= 0
2: i:=1
3: if I<= n goto 5
4: goto 17
5: t1:= I *3
6: t2:= a-3
7: t3:= t2[t1]
8: t4 := i*3
9: t5 := b – 3
10 :t6:=t5[t4]
11 :t7:=t3*t6
12 : t8:=p+t7
13 : p := t8
14 : t9 := i+ 1
15 : i := t9
16 : goto 3
17 :

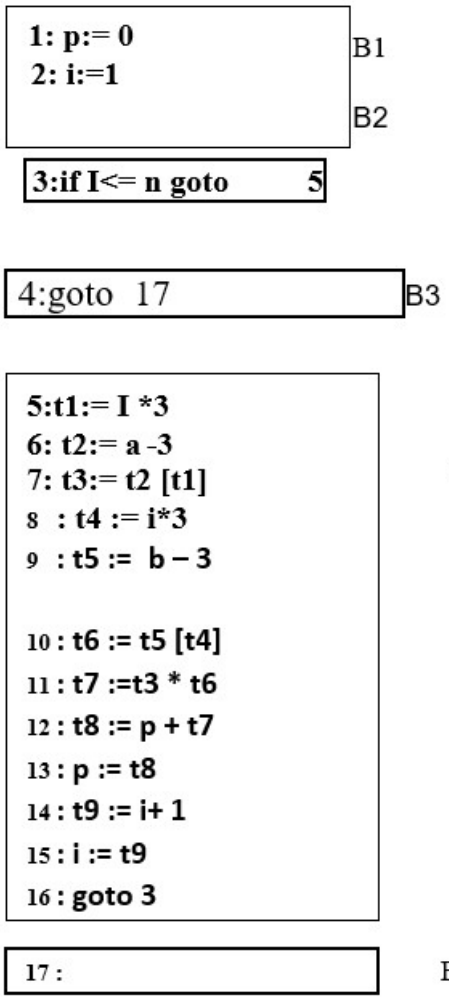
	op	Op1	Op2	Rés
1	:=	p	0	\
2	:=	i	1	\
3	<=	i	n	5
4	goto	\	\	17
5	*	i	$	$1
6	-	a	$1	$2
7	=[]	$1	$2	$3
8	*	i	$	$4
9	-	b	$	$5
10	=[]	$4	$5	$6
11	*	$3	$6	$7
12	+	p	$7	$8
13	:=	p	$8	
14	+	i	1	$9
15	:=	i	$9	
16	goto	\	\	3
17				

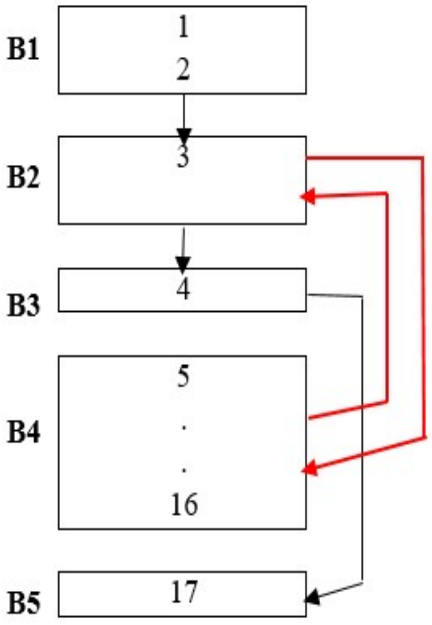
Les leaders de ce code sont les quadruplets dont les numéros sont : 1, 3, 4,5, et 17

6.2.2 Partitionnement en blocs fondamentaux

Un bloc fondamental est une séquence d'instructions débutant avec un leader et s'achevant à l'instruction qui précède le prochain leader.

En appliquant cette règle sur l'exemple précédent nous trouvons les blocs B1, B2 , B3,B4 et B5 suivants :





{B2, B4} forme une boucle

6.2.3 Construction du graphe de flow de contrôle

Les blocs fondamentaux représentent les nœuds du graphe de flow , il reste de créer les arcs du graphe. Un arc n'existera entre un bloc Bi et un bloc Bj que lorsque le leader de Bj est susceptible de suivre la dernière instruction de Bi au moment de l'exécution : Le graphe de flow obtenu pour notre exemple est donc :

- La structure du graphe de flow précédent est :

Bloc	Instructions	Préd	succ
B1	(1,2)	0	(B2)
B2	(3)	B1	B3,B4
B3	(4)	B2	B5
B4	(5…..16)	B2	B2

Tableau 9.3 : Construction du graphe de flow de contrôle

6.2.4 Localisation d'une boucle dans le graphe du flux

A partir de ce graphe nous allons détecter les boucles du code intermédiaire. Nous définissons une boucle comme étant un sous ensemble de l'ensemble X1,X2, …Xn des blocs fondamentaux tel que :

1) Pour toute paire (Xi, Xi+1) il existe un arc de Xi à Xi+1
2) Il existe un arc de Xn à X1
3) Il existe un prédécesseur de X1 autre que Xn

Dans le graphe de la figure précédente, la seule boucle qui existe est (B2 , B4).

Ecole normale supérieure de Constantine
Module : Compilation
Année universitaire 2023/2024

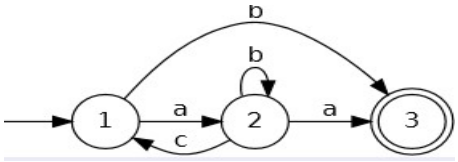
Département d'informatique et des sciences exactes
Section : info 3eme Année

TD 01

Exercice 01 :

Quelles chaînes sont reconnues par cet automate :

- aa
- acabcb
- acc
- abbc



Exercice 02 :

Ecrire l'automate fini non déterministe des expressions régulières suivantes :

- x|(yz)
- j*|kl
- ((a*|b*)c)*
- b(ab)*+(ba)*

Exercice 2' :

Décrire les langages dénotés par les expressions régulières suivantes :

- 1. a(a|b)*a
- 2. (("ε|a)*)*
- 3. (a|b)*a(a|b)(a|b)
- 4. a*ba*ba*ba*
- 5. (aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*

Exercice 03 :

Soient les expressions régulières suivantes :

(a*|b*)*, (a|b)*

Montrez quelles sont équivalentes ?

Exercice 04 :

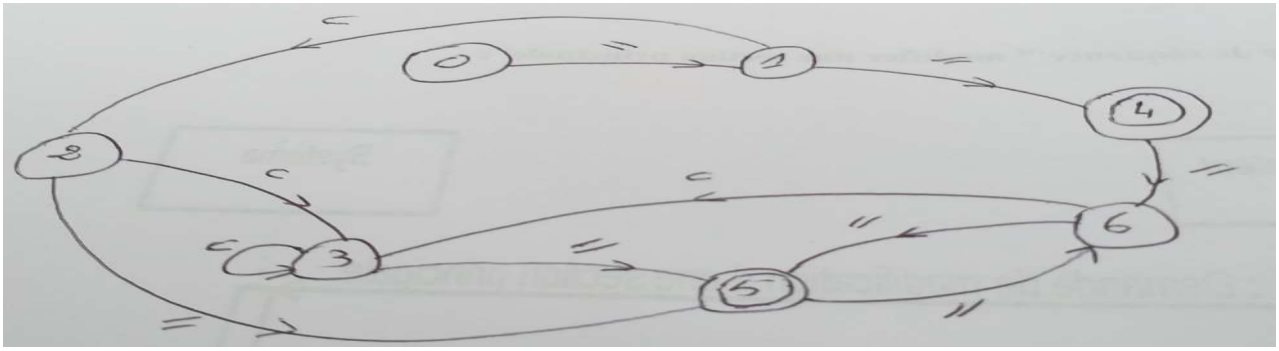
Soit le programme suivant :

```
Int max(i  j)
 Int i,j ;
/*compare  i and j*/
{Return(i>j ?i :j ;
```

- Identifiez et coder les tokens de ce programme ?
- Quels sont les erreurs détectées par l'analyseur lexical ?
- Quels sont les erreurs détectées par l'analyseur syntaxique ?

Exercice 05 :

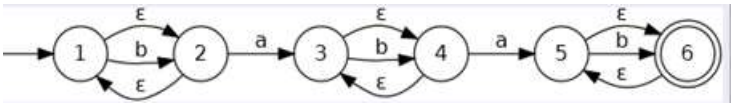
Soit l'automate suivant :

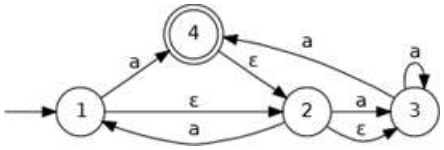


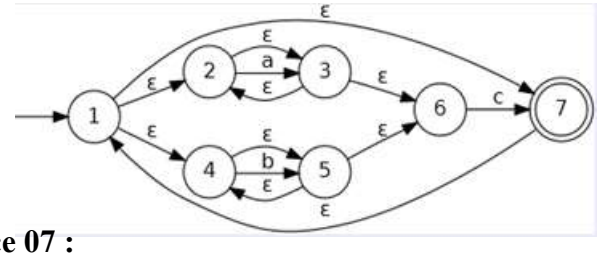
- Quelle est l'ER étudier à cet AFD ?
- Minimisez cet AFD

Exercice 06 :

Transformer en AFD les AFN suivantes :

- 

- 

- 

Exercice 07 :

Convertir les expressions régulières suivantes en des AFN puis en des AFD :

1. a(a|b)*a 2. (("ε|a)*)* 3. (a|b)*a(a|b)(a|b) 4. (a|b)*a(a|b)(a|b)(a|b)

5. a*ba*ba*ba* 6. (aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)* 7. aba*b*

8. (a|b)*aa(a|b)* (exam2016) 9. b*(abb*)*(a|ε) (examRAT2016) 10. ((xy*x) (yx*y))? (exam2017)

11. a((b|a*c))x*|x*a (examen 2018). 12. c(a*b|cd*)+|c (b*a|d*c) (examen rat. 2018)

Minimiser les AFD obtenus.

Ecole normale supérieure de Constantine

Module : Compilation

Année universitaire 2023/2024

Département d'informatique et des sciences exactes

Section : informatique 3eme Année

TD 02

Exercice 01 :

Soit la grammaire suivante :

R→R/R

R→R.R

R→R*

R→(R)

R→a

R→b

V_t= {/, . ,* ,(,) , a, b }

V_N={A}

- Cette grammaire est-elle ambiguë ? Justifier votre réponse ?
- Donnez l'arbre de dérivation correspondante à : (a*/b*).a*.b*/(a/b)*

Exercice 02 :

Soit la grammaire suivante :

E→E+T | T

T→T*F | F

F→x | y | (E)

- Donnez une grammaire équivalente à G qui soit LL(1) ?
- Construire la table prédictive pour la nouvelle grammaire ?
- Analyser la chaine : x+y*x

Exercice 03 :

Soit la grammaire suivante :

S→(L) | a

L→L,S | S

- Quel est le langage engendré par cette grammaire ?
- Construire la table prédictive pour cette grammaire ?

Exercice 04 :

Soit la grammaire suivante :

S → a|b|(T)

T → T,S|S

• Éliminer la récursivité et factoriser

• Déterminer les ensembles début et suivant

• Donner la table de l'analyse prédictive

• Donner le résultat de l'analyse prédictive de la chaîne w=(a,a),a,b donner l'arbre syntaxique

Ecole normale supérieure de Constantine
Module : Compilation
Année universitaire 2023/2024

Département d'informatique et des sciences exactes
Section : informatique 3eme Année

TD 03

Exercice 1 :

Soit la grammaire G dont les règles de production sont :

E → E or T | T

T → T and F | F

F → not F | (E) | true | false

1. Vérifier les conditions LL(1).
2. Supprimer la récursivité à gauche et factoriser.
3. Calculer les ensembles DEBUT et SUIVANT des non-terminaux de la nouvelle grammaire.
4. Donner la table de l'analyse prédictive de la nouvelle grammaire.
5. Donner la trace et le résultat d'analyse de la phrase : true and (false or true)

Exercice 2 :

Montrer que la grammaire suivante :

S → SA|A

A → a

est SLR mais pas LL(1)

Exercice 03 :

Soit la grammaire G dont les règles de production sont :

S → ABc| aSc

A → ε | aAB

B → bC | ε

C → Bc | aC | ε

1. Vérifier les conditions LL(1).
2. Supprimer la récursivité à gauche et factoriser.
3. Calculer les ensembles DEBUT et SUIVANT des non-terminaux de la nouvelle grammaire.

4. Donner la table de l'analyse prédictive de la nouvelle grammaire.

5. Donner la trace et le résultat d'analyse de la phrase : true and (false or true)

Exercice 04:

Soit la grammaire G suivante pour la gestion arborescente des fichiers et des répertoires :

- N = { H E REP FICH } où H est l'axiome,
- T = { nom [ ] }

H → H E | ε

E → REP | FIC

REP → nom [H]

FIC → nom

1. Donner la suite des dérivations droites dans G pour la chaîne w = rep1[fic1 rep2[ ]rep3[fic2] fic3] $
2. Calculer Début et Suivant pour les non terminaux de G.
3. Donner la table d'analyse LL(1) pour G.
4. La grammaire G est elle LL(1), pourquoi ?
5. Modifier la grammaire G pour quelle soit LL(1).
6. Donner l'Automate des items LR(0) canoniques pour G.
7. Donner la table des actions et successeurs SLR de G.
8. Donner la table de l'analyse ascendante dans G pour w = rep1 [fic1 rep2[ ] rep3 [fic2] fic3] $

Ecole normale supérieure de Constantine
Module : Compilation
Année universitaire 2023/2024

Département d'informatique et des sciences exactes
Section : informatique 3eme Année

TD 04

Exercice 01 :

Donner le code intermédiaire correspondant au bloc du code en C suivant : if (x < 200 && x!=y) x=0 ;

Exercice 02 :

On considère le bout de programme Pseudo-Pascal suivant :

```
neutre := 0;
abs := neutre;
opp := -x;
    while opp > x do
        begin
            tmp := x;
            x := opp;
            opp := tmp;+
        end;
abs := x;
```

- Donner la traduction en post fixé.
- Donner la traduction des trois premières instructions en quadruplet et en triplet.

Exercice 03 :

1. Donner l'arbre de syntaxe abstraite des instructions Pseudo Pascal suivantes :
   a) i := 1 + t[2 * x]
   b) t[3 * i] := 2 + x
   c) if c <= 2 then y := t [2] else f(4)
   d) if c <= 2 then t := new array of integer [2] else f(4)
2. Donner la traduction des trois instructions précédentes en quadruplet.

Exercice 04 :

Donner la traduction en post fixé et l'arbre de syntaxe abstraite des instructions Pseudo Pascal suivantes :
1. e := e + 1
3. while x <> y do if x > y then x := x - y else y := y - x

Exercice 05 :

Donner l'arbre de syntaxe abstraite des instructions Pseudo Pascal suivantes :
1. i := 0
2. t[i-1] := t[i+1]
3. while i < n do begin t[i] := readln(); i := i + 1 end

Exercice 06 :

Convertir l'expression a+-(b+c) en :

- Un arbre syntaxique.
- Quadruplets.
- Triplets.
- Triplets indirect.

Exercice 07 :

Soit le bloc du code en C suivant :

```
sum=0 ;
i=1 ;
l2 : if(i<=10)
{
    j=0 ;
il1 : if(j<1)
{
     sum +=j;
     j=j+I;
     goto l1;
}
ses = (i+1)*sum
i +=1;
goto l2;
}
```

Donner le code intermédiaire correspondant.

Ecole normale supérieure de Constantine
Module : Compilation
Année universitaire 2023/2024

Département d'informatique et des sciences exactes
Section : informatique 3eme Année

TD 05

Exercice 01 :

En utilisant les fonctions de génération de code pour machine à registre, écrivez la séquence de code produite pour les fragments de programme suivants :

```
x1 := 3; x2 := 3 + x1 + x2;
if not (x2 = x1) then
   x1 := x2
else
   x2 := 0
```

```
x2 : = 0;
while x3
   x3 := not x3 and (x2 = (x2 + 2));
```

```
var x,y:int
while not x=y do
      if x>y then x:=x-y else y:=y-x
done
```

Exercice 02 :

Soit le code suivant :

int x,y,z,a,b,c ;

x := a+b+c;

y := a*b + 2 *c ;

1. Générer du code 3 adresses et de l'assembleur en supposant un nombre non borné de registres.

2. Générer du code 3 adresses et de l'assembleur en supposant que l'on dispose uniquement de 4 registres : R1, R2, R3 et R4.

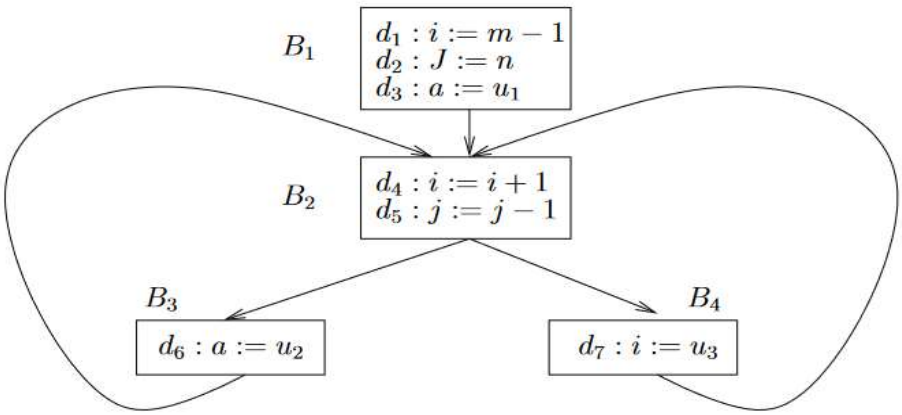
Exercice 03 :

Générer le graphe de flot de contrôle pour le code suivant :

```
while d>0 do{
     a:=b+c;
     d:=d-b;
     e:=a+f;
     if e>0
          {f:=a-d;b:=d+f}
     else
          {e:=a-c}
     b:=a+c}
```

Exercice 04 :

Après génération de code, on obtient le programme suivant :



Supprimer le code mort.

Références bibliographiques

1. Compilateurs : Principes, techniques et outils, auteurs : Aho, Ullman & Sethi, Edition : Intereditions.
2. Principles of compiler design, auteurs : Aho & Ullman, Edition Addison Vesley
3. Alfred Aho, Ravi Sethi, and Jeffrey Ullman. Compilateurs. Principes, techniques et outils. Dunod, Paris,1991
4. Alfred Aho and Jeffrey Ullman. The Theory of Parsing, Translating and Compiling. Prentice Hall Inc, 1972.
5. Andrew Appel. Modern Compiler Implementation in C. Cambridge University Press, 1998.
6. [John Levine, Tony Mason, and Doug Brown. Lex & yacc. O'Reilly & Associates, Inc., 1990.
7. AHO A., Ravi SETHI R., ULLMAN.J. “ Compilation : principes, techniques et outils”, Addison-Wesley.
8. Develay I. « la compilation », ISTIA 2004
9. Di Cosmo R. « cours de compilation », Paris VII 2006
10. Risset T., « cours de compilation »,2004
11.
12. https://www.tutorialspoint.com/compiler_design/compiler_design_syntax_analysis.htm
13. https://eboik.com/cours-compilation-pdf/