TRAITEMENT DES LANGAGES ET COMPILATEURS

M^me BAGNY-TCHEDRE A. P.
Ing. Informatique

Edition du 25/06/2021

Sommaire

Chapitre 1 INTRODUCTION 3
Chapitre 2 ANALYSE LEXICALE 15
Chapitre 3 LES AUTOMATES A ETATS FINIS 20
Chapitre 4 ANALYSE SYNTAXIQUE 29
Chapitre 5 TYPAGE 32
Chapitre 6 ANALYSE SEMANTIQUE 40

Chapitre 1 INTRODUCTION

1- Concepts de base dans la conception et la construction d'un compilateur

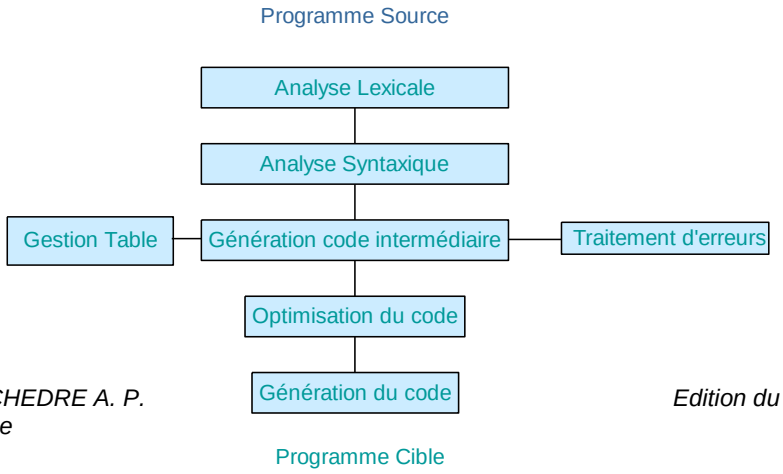
Les concepts de base sont les suivantes :

- **Langage Source** : langage dans lequel les **programmes acceptés** par le compilateur sont écrits.
- **Langage Cible** : langage dans lequel le langage Source est compilé.
- **Langage d'implémentation :** langage dans lequel le **compilateur** lui même est écrit.
- **Compilateur à une passe** : le texte du programme source est traité en *une fois*.
- **Compilateur à plusieurs passes** : un certain nombre de représentations du Programme Source est construit par le compilateur. Chaque représentation est obtenue à partir de la précédente en une *passe.*
- **Interpréteur** : de manière intrinsèque, simulateur d'un ordinateur. Usuellement, désigne un compilateur qui génère **le code d'une machine abstraite** en tant que **langage cible**, puis appelle l'interpréteur de ce langage.
- **Compilateur Load-and-go** : une autre désignation de l'interpréteur. Dans certains cas, les compilateurs Load-and-go **génèrent le langage machine** avec des **adresses absolues** (pas d'éditeur de liens).

Remarques :

- Les compilateurs Load-and-go sont généralement beaucoup plus faciles à développer, et leur vitesse de compilation est élevée. Cependant, ils ne permettent pas l'intégration de programmes sources utilisateur avec des programmes dans d'autres langages, ni avec les bibliothèques.
- L'intégration d'un compilateur avec un éditeur de liens existant et le système de fichiers n'est pas une tâche triviale. Elle est comparable au développement du compilateur.
- Le développement d'un compilateur multi-passes peut être plus facile à gérer que celui du mono-passe.
- L'efficience du compilateur et celle du code objet sont généralement des aspects complémentaires à la compilation : plus le code généré par le compilateur est efficient plus le niveau de l'optimisation est élevé et, par conséquent, moins le compilateur est efficient.
-

2-Phases de base d'un compilateur



- **L'analyseur Lexical** (scanner) traduit une **séquence de caractères** qui représente le programme source en une séquence de ***symboles*** *(tokens);*
- **L'analyseur syntaxique** (parser) vérifie **la conformité** de la séquence de symboles avec **la grammaire du langage source**, et produit **un arbre syntaxique** représentant le programme source;
- **Le générateur de code intermédiaire** convertit l'arbre syntaxique augmenté de l'information sémantique en code de la machine abstraite;
- **L'optimisation du code** augmente l'efficience du code intermédiaire en appliquant un ensemble de transformations de code prédéfinies, indépendantes de la machine;
- **La génération du code** génère **le langage cible**.

Les données expérimentales montrent que :

- l'analyse lexicale consomme approximativement 10% du temps de compilation;
- la génération de code compte pour 50 à 70% du temps de compilation
- le recouvrement d'erreurs prend les 20 à 40% restant

3- L 'Analyse Lexicale

Elle n'est pas indispensable (l 'Arbre Syntaxique peut lire des caractères individuels) mais utile car elle aide dans la réduction de la complexité de l 'Arbre Syntaxique et augmente sa portabilité (si la lecture au niveau caractère est dépendante de la machine).
Le texte d'entrée (commentaires, indentation …) sont transformés en une séquence de tokens, typiquement par une procédure qui retourne un token à la fois.
Les services additionnels sont :

- construction des tables de symboles
- Book keeping (ex numérotation des lignes dans le listing source)

3-1 Tokens et Lexèmes

Un token est un **symbole** retourné par l 'Arbre Syntaxique Lexical tandis qu'un lexème est **une séquence de caractères d'entrée** correspondant à un seul token.

Exemples de lexèmes: identificateur, nombre, keyword, string, opérateur (ou quelques classes: operateur arithmétique, relationnel), ponctuation.

L'Arbre Syntaxique est souvent intéressé par la classe et non le lexème lui même par exemple les contenus d'une chaîne ne sont pas importants pour lui.
Un token peut être représenté par une paire : **classe - lexème**
Certaines classes ont un seul lexème

Ex : **classe** « keywordIF » - **lexème** if

Les classes ayant plusieurs lexèmes requièrent des patrons pour savoir si le lexème actuel est valide.

Ex de patron (informel)

« un identificateur est une séquence de lettres et de chiffres commençant par une lettre »
Un lexème est valide s'il correspond au patron.

Attributs des tokens :

- Pour un nombre: sa valeur
- Pour un identificateur: le type, le niveau d'imbrication etc.
- Pour un string: son contenu

Ces attributs sont affectés aux tokens par l'analyseur lexical, et stockés dans la table des symboles. L'analyseur lexical a accès à eux et peut les retourner à l'analyseur syntaxique à la demande. Ceci réduit le volume de tests pour l'analyseur syntaxique.

Spécification des tokens

Les lexèmes sont représentés par des chaînes de caractères (ou des chaînes construites sur des alphabets plus réduits, tels que les lettres).
Ces chaînes ont généralement une structure simple et régulière. Un ensemble de toutes les chaînes valides est un langage.
Nous avons besoin d'une description concise du langage (même s'il est infini, comme c'est le cas par exemple pour l'ensemble de tous les programmes possibles!).
Une grammaire, ou un certain mécanisme/formalisme équivalent, constitue une telle description concise.

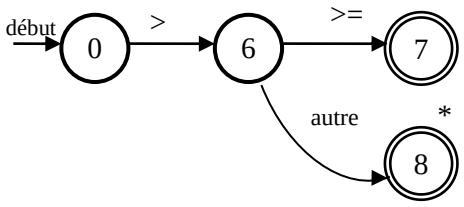
Reconnaissance des Tokens

Les expressions Régulières implique le **graphe de transition** et **l'analyseur Lexical**
Les Réseaux de transition décrivent les actions qui ont lieu lorsque l 'Arbre Lexical est appelé par l'Arbre Syntaxique pour délivrer le token suivant. Ils sont formés d'états et d'arcs.

Etat : position courante dans le lexème;

Arc : étiqueté par un symbole de l'alphabet: le symbole (courant) possible.

Le réseau qui suit peut être utilisé pour reconnaître 2 tokens: > et >=. La reconnaissance s'arrête si on atteint un état dans un double cercle. Dans l'état marqué par un *, on doit remettre un caractère dans la chaîne d'entrée : il n'a pas été consommé mais juste examiné.



5-L'analyseur lexical

Au niveau de l'Analyseur lexical (scanner), les flots de caractères constituant le programme source sont lus de gauche à droite, groupés en unités lexicales et les caractères superflus supprimés. Une unité lexicale (token) est une suite de caractères ayant une signification collective.
Le but essentiel de l'analyse lexicale est de :

- générer des unités lexicales
- déterminer si chaque unité lexicale est un mot du vocabulaire du langage utilisé

Exemple : Instruction en Pascal **vitesse := vitesse_initiale +accélération*40**

Le scanner regroupe les caractères dans les unités suivantes :

Identificateur	vitesse
Symbole d'affectation	:=
Identificateur	vitesse
Symbole d'addition	+
Identificateur	accélération
Symbole de multiplication	*
Nombre	40

Si l'analyseur lexical rencontre un mot non autorisé alors il génère une erreur.
Exemple : Si $ n'est pas utilisé dans le langage considéré alors un message d'erreur est généré lorsque $ est rencontré
Remarque : Le scanner ne travaille pas sur plusieurs unités lexicales à la fois
Exemple : Langage Pascal
Soit la chaîne de caractères := +*kk-45
Cette chaîne est jugée correcte par le scanner qui reconnaît les unités lexicales suivantes, (correctes lorsqu'on les considère séparément).

Analogie : Langue française
Soit la phrase : tu manger des cailloux
Cette phrase n'est évidemment pas correcte grammaticalement, mais le vocabulaire qu'elle contient est correct. Une analyse lexicale n'y détecte donc aucune erreur.

6- L'analyseur syntaxique (parser)

Le but du Parser est de:
- regrouper **les unités lexicales** en structures grammaticales **correctes**
- déterminer si **la syntaxe** (ou GRAMMAIRE) est **correcte**

La grammaire d'un langage est définie par un ensemble de règles.

Exemple : Soit l'instruction **vitesse := vitesse_initiale + accélération*40**

Les règles qu' 'on peut utiliser sont les suivantes :

Pour la définition des expressions :

Règle 1 : Tout *identificateur* est une expression
Règle 2 : Tout *nombre* est une expression
Règle 3 : Si *expression1* et *expression2* sont des expressions, alors :

expression1 + expression2
*expression1 * expression2* ⟶ sont des expressions
(*expression1*)

Pour la définition d'une instruction

Règle 4 : *identificateur* := *expression* est une instruction

Règle 1 implique :

vitesse
vitesse_initiale ⟶ sont des expressions
accélération

Règle 2 implique :

40 est une expression

Règle 3 implique :

*accélération * 40* sont des expressions
*vitesse_initiale + accélération * 40*

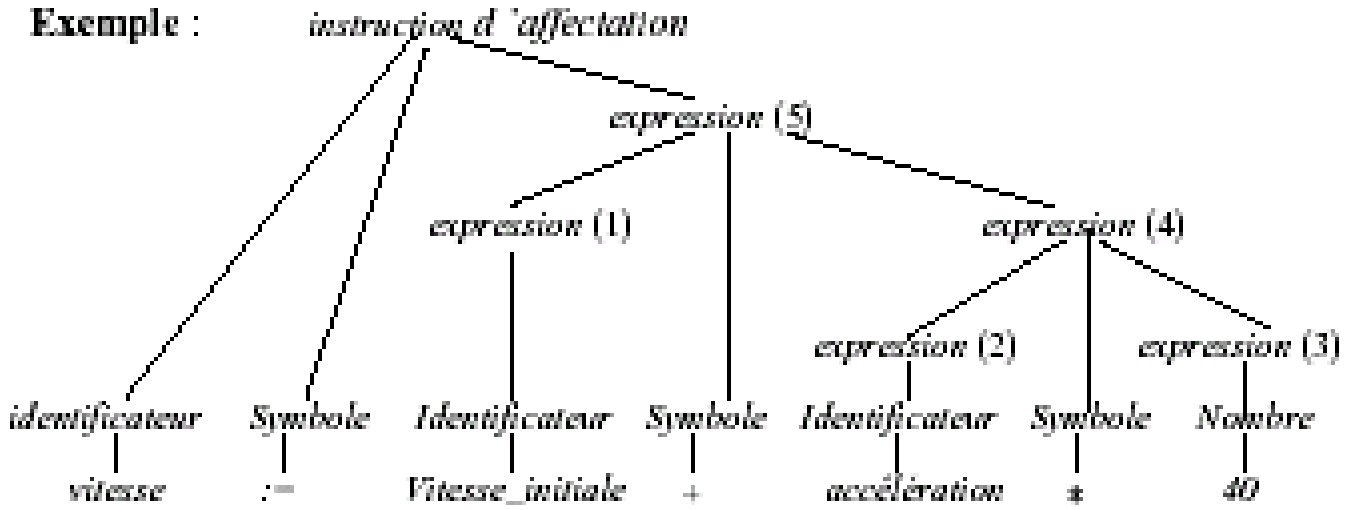
(On suppose ici qu' 'il y a précédence de * sur +)

Règle 4 implique :

***vitesse* := *vitesse_initiale* + *accélération**40** est une instruction

L'analyse syntaxique peut être représentée par un arbre syntaxique de dérivation

Exemple :



L'analyseur lexical construit les feuilles de l'arbre, qui sont des unités lexicales
L'analyseur syntaxique construit :

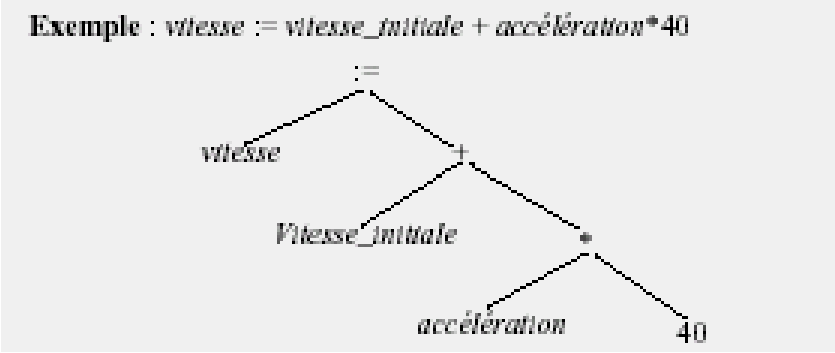
- les expressions 1 et 2 à l'aide de la règle 1
- l'expression 3 à l'aide de la règle 2
- les expressions 4 et 5 à l'aide de la règle 3

Si l'analyseur syntaxique rencontre une structure qui n'est pas définie par les règles de la grammaire, alors il signale une erreur.

Exemple : *vitesse* := *vitesse_initiale* +) *accélération**40

- La règle 4 définit une instruction d'affectation telle que le membre à droite de := est une expression
- Les règles 1, 2 et 3 ne permettent pas de construire la partie à droite de :=.

Le parser détecte alors une erreur. Cela se traduit par le fait qu'on ne peut pas construire d'arbre syntaxique de dérivation.
Il existe une représentation des structures des phrases qui est plus concise que les arbres syntaxiques de dérivation. Il s'agit des **arbres syntaxiques abstraits** (en anglais : Abstract Syntax Tree, AST).
Dans un AST, les opérateurs sont nœuds et les opérandes sont des feuilles



7- L'analyseur sémantique (contextuel)

Son but essentiel est de déterminer si le programme contient des erreurs sémantiques statiques. L'analyseur sémantique vérifie, entre autres :
Les types des identificateurs :
Il détecte par exemple des erreurs du genre :

- opération sur deux types incompatibles
  Exemple : ensemble + fichier
  fonction * tableau
- nombre réel utilisé comme indice d'un tableau. Exemple : tab[2.3]

Il peut aussi effectuer certaines « corrections » sur des types de variables.
Exemple : entier + réel .L'entier est converti en réel avant d'effectuer l'addition.

La portée des identificateurs :
Une erreur est détectée lorsque, par exemple, une variable est d 'une part, déclarée à 'intérieur d 'une procédure et d 'autre part, utilisée à l 'extérieur de la procédure.

Unicité des identificateurs :
Exemples :
- Une variable doit être déclarée une seule fois (dans une même portée)
- Les étiquettes dans une instruction case (switch) doivent être distincts
- Les éléments d'un type *énumération* ne peuvent pas être répétés.

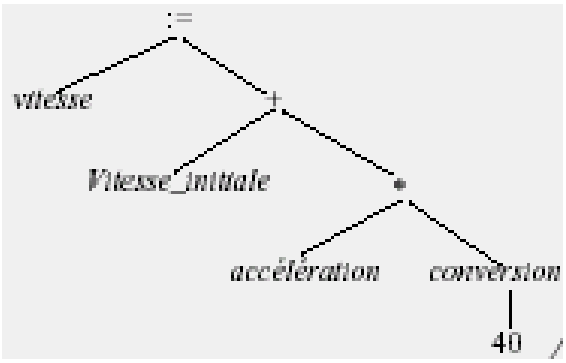
Flot d'exécution :
Exemple : Une instruction *break* termine l'exécution de l'instruction *while, for* ou *switch* qui l'englobe au plus près. Il y a erreur si une telle instruction (*while, for* ou *switch*) n'existe pas alors qu' 'une instruction *break* est utilisée.
Parmi les différents contrôles effectués, le contrôle de type est le plus complexe.

Exemple de contrôle de type :

***vitesse* := *vitesse_initiale* + *accélération**40**

Supposons que *vitesse*, *vitesse_initiale* et *accélération* sont des réels, l'analyseur sémantique (contrôleur de type) :
- Le parser génère un arbre syntaxique abstrait (AST détecte multiplication entre réel (accélération) et nombre entier (40)
- convertit nombre 40 en un réel 40.0) qui est ensuite complété par l'analyseur sémantique.

Pour notre exemple, le parser génère l'AST et le contrôleur de type (type checker) y insère une conversion pour obtenir l'AST suivant :



8- Génération de code intermédiaire

Après l'analyse sémantique, certains compilateurs produisent une représentation intermédiaire du code source. Celle-ci doit :
- être indépendante de la machine cible (contrairement aux langages assembleurs)
- décrire en détail les séquences d'opérations à effectuer (comparable à l 'assembleur)

C'est au fait un code exécutable d'une machine abstraite. Ce code doit être :
- d'un côté, facile à produire,
- d'un autre, facile à traduire en langage cible.

Exemples de représentations intermédiaires :
Exemple 1 : AST
Exemple 2 : Code à trois adresses (en anglais : **Three-Adress Code, TAC**)

Dans chaque instruction, il y a :
- **au plus un** opérateur (binaire ou unaire)
- **au plus trois** opérandes
- **une affectation**

Les variables temporaires utilisées pour stocker les valeurs calculées.
Par exemple, pour l'instruction : ***vitesse* := *vitesse_initiale* + *accélération**40**
Nous obtenons l'AST vu plus haut et le TAC suivant :

t1 :=*conversion*(40)
t2 := *accélérationt1**
t3 :=*vitesse_initiale* + t2
***vitesse* := t3**

9- Optimisation de code

Elle consite en l'amélioration du code intermédiaire selon les critères suivants :

- le temps : durée d'exécution à minimiser
- la mémoire : occupation de la mémoire à minimiser

Les deux critères étant incompatibles, il faut faire des compromis :
Exemple : **vitesse := vitesse_initiale + accélération*40**
La conversion de 40 en un réel peut être effectuée une fois pour toutes, lors de la compilation.
Les instructions

t1 :=*conversion*(40)

deviennent **t1 :=*accélération**40.0**

t2 :=*accélérationt1**

Considérons les deux instructions

t3 *:=vitesse_initiale* + t2
***vitesse :=*t3**

Si ultérieurement aucune opération n'est effectuée sur t3, alors les 2 instructions peuvent être remplacées par : **vitesse := vitesse_initiale + t2**
Les 4 instructions TAC de la page précédente deviennent alors :

t1 *:=accélération*40.0*
***vitesse := vitesse_initiale* + t1**

10- Production de code

Le code cible est en langage assembleur ou machine. Le code engendré peut être exécutable ou nécessiter une édition de liens avec des librairies.
Cette phase dépend directement de la machine sur laquelle tournera le programme
En effet, ce code est constitué d'instructions exécutables par le microprocesseur de la machine cible et utilise des registres dont dispose la machine cible
Exemples :
- Intel 80386, 80486, …
- Motorola 68020, 68030 ...

11- Parties frontale et finale

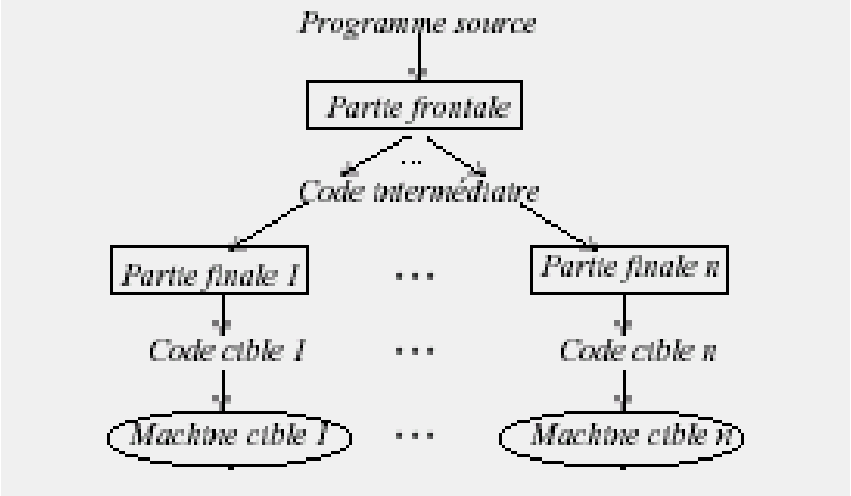
Les phases d'un compilateur peuvent être regroupées en deux parties:
La partie frontale qui contient les cinq phases suivantes :

- analyse lexicale
- analyse syntaxique
- analyse sémantique
- génération de code intermédiaire

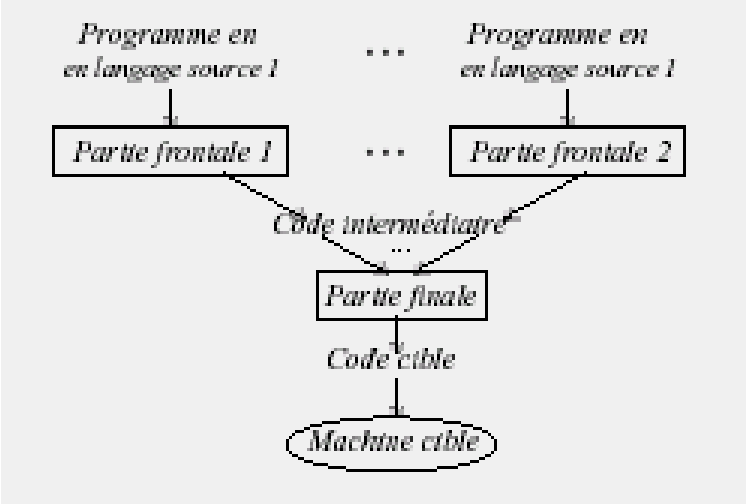
- optimisation de code

Cette partie dépend du **code source** et du **code intermédiaire** mais ne dépend pas de la machine cible.
Pour sa conception, on ne se soucie pas des détails de la machine sur laquelle le programme est sensé tourner.
La Partie finale consiste en la génération de code cible. Cette partie ne dépend pas du code source mais dépend du code intermédiaire et de la machine cible.
Pour sa conception, on ne se soucie pas du langage source utilisé.

Les avantages à effectuer séparation en deux parties sont :
Premier avantage : lorsqu'on change de machine, seule la partie finale doit être modifiée

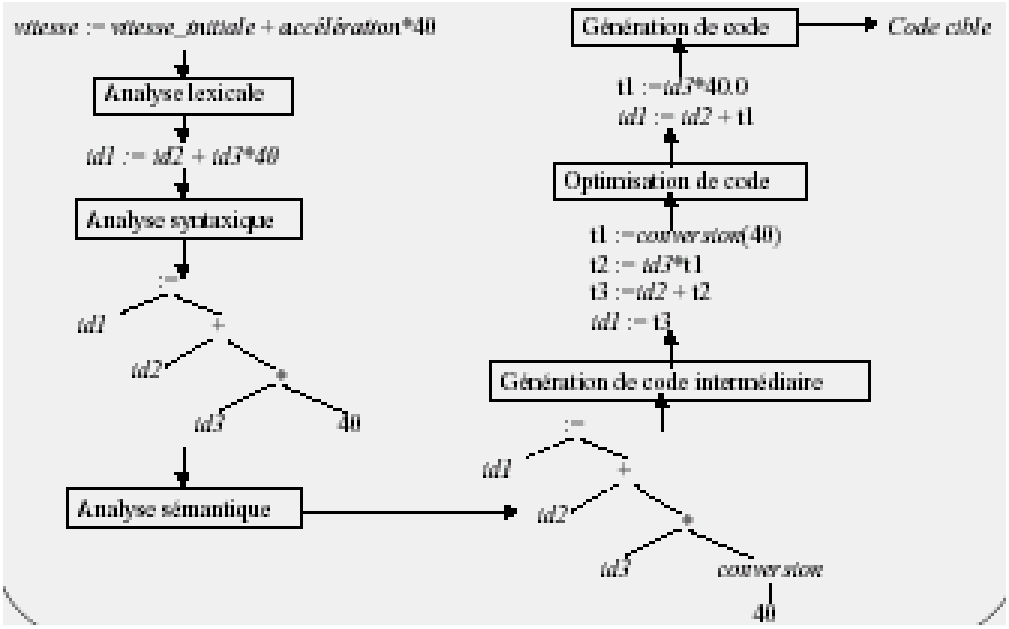


Second avantage : lorsqu'on change de langage source, seule la partie frontale doit être modifiée.



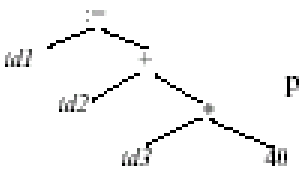
Récapitulation des différentes phases pour l'instruction :

***vitesse* := *vitesse_initiale* + *accélération**40**

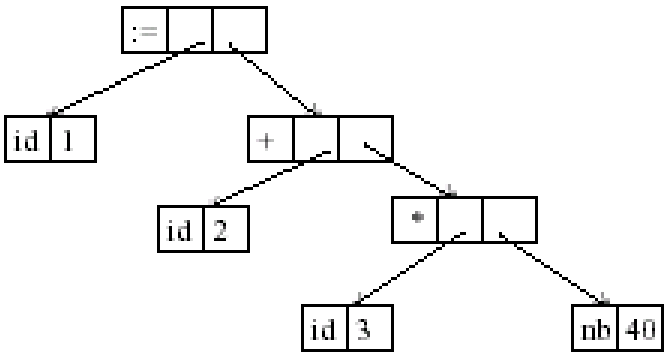


Implémentation de la structure de données engendrée par l'analyseur syntaxique

L 'AST



peut être implanté comme suit :



12- Nombre de passes dans un compilateur

Une seule passe : Les différentes phases ne sont pas séparées et il n'y a pas moyen d'avoir accès aux résultats intermédiaires :

- entrée : programme en code source
- sortie (résultat) : programme en code cible

Code source ⟶ *Compilation* ⟶ *Code cible*

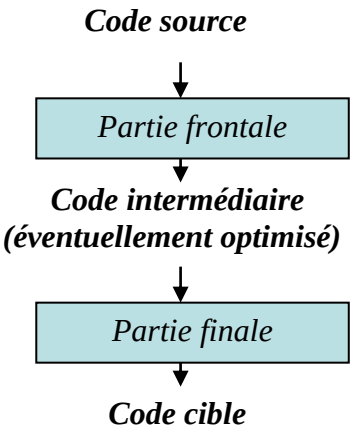
Une passe pour chaque phase :

- Les différentes phases sont séparées
- Le résultat d'une phase est une entrée pour la phase suivante

Avantage : « Diviser pour régner »
Chaque phase est relativement facile à mettre au point

On peut aussi avoir plusieurs passes, et plusieurs phases par passe.
Exemple :

- une passe pour la partie frontale
- une passe pour la partie finale



13- Réaliser un compilateur

La réalisation d'un compilateur est une **tâche complexe.** Il est donc préférable d'utiliser un **langage de haut niveau**. Exemple : C, C++, Ada

Que faire si on ne dispose pas d'un langage de haut niveau ?
On procède à l'analyse suivante :

- Soit **L** le langage pour lequel on veut **écrire un compilateur**
- Soit **A** le langage assembleur de la **machine cible**
- Soit **M** le langage assembleur du système de développement utilisé

<u>Étapes pour réaliser un compilateur</u> **:**

1 : On détermine un sous-ensemble L' de L qui est suffisant pour écrire un compilateur de L

2 : On écrit un compilateur de L'en langage assembleur M

3 : On écrit un compilateur de L en langage L'

Exemple :

- On développe, sur une station de travail SPARC, un compilateur Pascal pour PC
- Le seul langage de programmation disponible est l'assembleur SPARC
- L'assembleur exécutable sur PC est le 80486 d'Intel

On a donc : L = Pascal,
A = Assembleur 80486 pour Intel, et
M = Assembleur SPARC

Étape 1 : On écrit en Pascal le code source du compilateur Pascal pour PC. À partir du code source de ce compilateur, on détermine un sous-ensemble L' de Pascal qui est suffisant pour l'écriture du compilateur.
Exemple : Si le code source du compilateur ne contient pas la structure *repeat*, alors L' n'accepte pas *repeat.*
Étape 2 : On écrit en langage assembleur SPARC un compilateur L' pour PC
Ce compilateur traduit un programme écrit en L'en un programme écrit en assembleur 80486
Étape 3 : Le code source du compilateur écrit dans ***l'étape 1*** est compilé à l'aide du compilateur écrit dans ***l'étape 2***.

Chapitre 2 ANALYSE LEXICALE

1- Spécification des tokens

Un ***lexème*** est une séquence de caractères et un token est un ***token*** est une catégorie de *lexèmes* définis par des *patterns*.

Exemple: resultat4 = nx4a * 4.25;

Un Token peut être un :

Identificateurs: lettre suivie de lettres et digits
Nombre: débute par un chiffre suivis de chiffres peut inclure un '.' suivi de chiffres
Opérateur: +, -, *, /
Séparateur: ;

2- Définitions

Un ***Langage*** est un ensemble de chaînes sur un alphabet donné.
Un ***Alphabet*** est un ensemble fini de symboles.

Exemples:

- {0,1} → alphabet binaire
- {a,b,...,z}
- caractères ASCII, EBCDIC, UNICODE

Une ***Chaîne*** est une séquence finie de symboles de l'alphabet.

- longueur d'une chaîne |s|
- chaîne vide ε
- c*oncaténation de chaînes* x et y → xy
  ⇒ chaîne vide ε élément identité

- exponentiation
  $s^0 = \varepsilon$
  $s^1 = s$
  $s^i = s^{i-1}s$ (pour i>0)
- préfixe de *s*
  chaîne obtenue en supprimant 0 ou plusieurs symboles à la fin de s
- suffixe de *s*
  chaîne obtenue en supprimant 0 ou plusieurs symboles au début de s
- sous-chaîne de *s*
  chaîne obtenue en supprimant un préfixe et un suffixe de s
- préfixe, suffixe, sous-chaîne *propre* de *s*
  préfixe, suffixe, sous-chaîne de *sx*, tel que $s \neq x$
- sous-séquence de s
  chaîne obtenue en supprimant des symboles (pouvant être non contigus) de s

3- Opérations sur les langages

Soient L et M 2 langages:

- L ∪ M **union de *L* et *M*** : {s | s élément de *L* ou de *M*}
  {abc, bb} ∪ {c, cd, cdd} = {abc, bb, c, cd, cdd}

- LM **concaténation de *L* et *M*** : {st | s élément de *L* et t élément de *M*}
  {abc, bb}{c, cd, cdd} = {abcc, abccd, abccdd, bbc, bbcd, bbcdd}
- L* **fermeture** ou **clôture** *de Kleene* de *L* : 0 ou plusieurs concaténations de *L*
  {ab, c}* ={ε, ab, c, abab, abc, cab, cc, abababab, ababc,abcab,abcc, cabab, cabc, ccab,ccc, ...}
- L+ **fermeture** *ou* **clôture** *positive* de *L*: 1 ou plusieurs concaténations de *L* ou la clôture sans la chaîne vide ε

Exemples avec : **L = {A, ..., Z, a, ..., z}** et
D = {0, ..., 9}

- Le langage **Entiers = D+** est formé de toutes les chaînes constituées de un ou plusieurs chiffres (notons qu'il n'y a aucune restriction sur de tels entiers).
- Le langage **Octal = {0, 1, 2, 3, 4, 5, 6, 7} +** est l'ensemble de tous les entiers sans signe représentés en Octal.
- Le langage **Id = L (L ∪ D)*** est l'ensemble de tous les identificateurs possibles (sans restriction de longueur).
- Le langage **Nombres = ({+, -} ∪ {ε}) (D+) ({ε} ∪ ({.} (D+)))** contient les nombres entiers et réels avec un signe optionnel.

4- Expressions Régulières

Il s'agit d'un formalisme plus simple pour la description des langages réguliers. Chaque expression régulière *r* dénote un langage *L(r).*
Ce formalisme peut être présenté avec plusieurs variations, mais l'essence est la même.

- {ε} → ε
- {x} → x x appartenant à l'alphabet

- **Si** L1 → e1 et L2 → e2 **Alors**
  L1∪ L2 → e1| e2
  L1 L2 → e1e2
  L1* → e1*
  L1+ → e1+

Les expressions régulières peuvent être construites à partir d'autres expressions régulières en appliquant *des **règles syntaxiques***.

Règles syntaxiques

Etant donné un alphabet **Σ**:

1. **ε** est une expression régulière
   ⇒ dénote le langage {**ε**}
2. pour tout symbôle **a** de **Σ**, **a** est une expression régulière
   ⇒ dénote le langage {**a**}

3. si r et s sont deux expressions régulières
   1. (r)|(s) est une expression régulière
      $\Rightarrow$ dénote le langage $L(r) \cup L(s)$
   2. (r)(s) est une expression régulière
      $\Rightarrow$ dénote le langage $L(r)L(s)$
   3. (r)* est une expression régulière
      $\Rightarrow$ dénote le langage $(L(r))^*$
   4. (r) est une expression régulière
      $\Rightarrow$ dénote le langage $L(r)$

Précédences

1) *, associativité gauche
2) concaténation, associativité gauche
3) |, associativité gauche

Exemple: $\Sigma=\{a,b\}$

1. a|b $\Rightarrow$ {a,b}
2. (a|b)(a|b) $\Rightarrow$ {aa, ab, ba, bb}
   même que aa|ab|ba|bb
3. a* $\Rightarrow$ chaînes de 0 ou plusieurs a {ε,a,aa,...}
4. (a|b) $\Rightarrow$ chaînes de 0 ou plusieurs a ou b
   même que (a*b*)*
5. a|a*b $\Rightarrow$ chaîne a et chaînes formées de 0 ou plusieurs a suivis d'un b

Propriétés des expressions régulières

Axiome	*Description*
r|s = s|r	**commutativité de |**
r|(s|t) = (r|s)|t	**associativité de |**
(rs)t = r(st)	**associativité de concaténation**
r(s|t) = rs|rt	**distributivité de concaténation**
(s|t)r = sr|tr	**par rapport à |**
ε r = r	**élément identité**
r ε = r	**pour la concaténation**
r* = (r| ε)*	**relation entre * et ε**
r = r***	**idempotence de ***

Définitions régulières

Soit un alphabet Σ,
Une définition régulière est une séquence de définitions de la forme
$d_1 \rightarrow r_1$
$d_2 \rightarrow r_2$
…..
…
$d_n \rightarrow r_n$

Les d_i sont des noms distincts

Les r_i sont des expressions régulières construites à partir des symboles de Σ et des noms dj tels que j<i

Exemple: identificateurs
lettre → A|B|...|Z|a|b|...|z
digit → 0|1|...|9
id → *lettre* (*lettre*|*digit*)*

Abréviations

Pour faciliter l'écriture de constructions fréquentes
1. *r+* : 1 ou plusieurs instances de r (*r+ = rr**)
2. *r?* : 0 ou 1 instances r (*r? = r|ε*)
3. classes de caractères: *[abc] = a|b|c*
   [a-z] = a|b|...|z

Il est, aussi, possible d'utiliser des parenthèses pour grouper les expressions régulières.
Exemples :

(0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)+ signifie toute séquence non vide de chiffres.
Une abréviation de ce qui précède est l'intervalle : [0-9] +

Une autre abréviation peut être utilisée pour « zéro ou une occurrence », c'est à dire pour quelque chose d'optionnel : au lieu de **(+ | - | ε)** on peut écrire **(+ | -) ?**

Limitation des Expressions régulières
Certains langages ne peuvent pas être représentés par des expressions régulières.
Exemples:
- Constructions balancées ou incluses
  Langage de parenthèses balancées
  {wcw|w est une chaîne de a et b}
- Langage où on considère un nombre fixe de répétitions

5- LEX

C'est un outil pour l'analyse lexicale. Il prend en entrée des expressions régulières et génère du code qui reconnait ces expressions régulières.

Forme des spécifications Lex

declarations
%%
productions
%%
code additionnel

- **Déclaration**: déclaration de variables, constantes, expressions régulières
- **Productions**: spécification d'actions à exécuter lorsque les expressions régulières sont reconnues.

expression_reguliere action

- **Code additionnel**: fonctions additionnelles

Exemple LEX

```
        int nb_lignes= 0, nb_cars= 0;
    %%
    \n ++nb_lignes; ++nb_cars;
    . ++nb_cars;
    %%
    main()
    {
    yylex();
    printf("nomdre de lignes = %d,
    nomdre de caractères = %d\n",nb_lignes, nb_cars);
    }


{%
#include <stdio.h>
%}
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
%%
{DIGIT}+ {
        printf("Un Entier: %s (%d)\n", yytext, atoi(yytext));
       }
{DIGIT}+"."{DIGIT}*   { printf( "Un float: %s (%g)\n", yytext, atof(yytext));  }
if|then|begin|end|procedure|function       { printf("Un keyword: %s\n", yytext); }


{ID}      printf("Un identificateur: %s\n", yytext);
"+"|"-"|"*"|"/" printf(" Un opérateur: %s\n", yytext);
"{"[^}\n]*"}"   /*supprimer une ligne de commentaires */
[ \t\n]+        /* supprimer espace */
.       printf("caractère inconnu: %s\n", yytext);

%%
int main(int argc, char *argv[]){
  ++argv, --argc;  /* sauter nom du programme */
  if (argc > 0)
     yyin = fopen(argv[0], "r");
     else
       yyin = stdin;
     yylex();
}
```

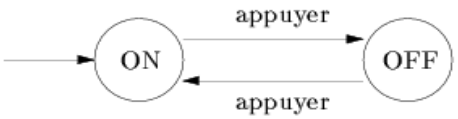
Chapitre 3 LES AUTOMATES A ETATS FINIS

On peut formaliser la spécification d'un langage selon le type du langage :

- Pour un langage fini on procède à une énumération exhaustive des mots
- Pour un langage infini, on peut utiliser des :
  - Automates à états finis
  - Expressions régulières
  - Grammaires
  - Machines de Turing

1- Automates à états finis (AEF)

- Un modèle d'un système et de son évolution, c'est à dire une description formelle du système et de la manière dont il se comporte.
- Les automates sont des modèles très simples et ne sont évidemment pas en mesure de décrire tous les systèmes que l'on est amené à modéliser.
- Un AF est composé d'un ensemble fini d'états (représentés graphiquement par des cercles), d'une fonction de transition décrivant l'action qui permet de passer d'un état à l'autre (ce sont les flèches) et d'un état initial (l'état que désigne la flèche isolée). Un AF est donc un graphe orienté où les nœuds correspondent aux états et les arcs aux flèches.
- Exemple : Un bouton poussoir passant alternativement d'un état «ON» à un état «OFF» par l'action d'appuyer sur le bouton peut être représentée par:



Formulation mathématique d'un AEF

- Un automate à états finis est un 4-uplet <Q, Σ, δ, q0>
  - Q ensemble fini des états de l'automate
  - Σ l'alphabet (ensemble fini des symboles)
  - δ une fonction de transition $Q \times \Sigma \rightarrow Q$ exprimant le passage entre deux états
  - q0 l'état initial
- Fonction $\delta^* : Q \times \Sigma^* \rightarrow Q$ extension de δ
  - $\delta^*(q,\varepsilon)=q$
  - $\delta^*(q,xy)=\delta(\delta^*(q,x),y)$

Accepteur de langages

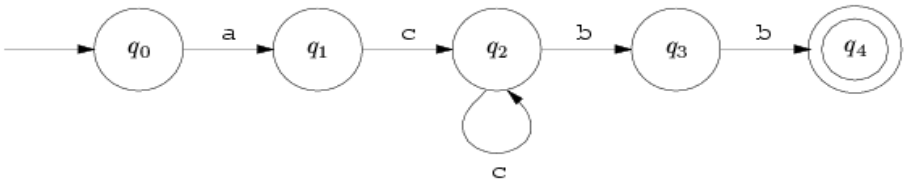
Les automates finis peuvent être considérés comme des modèles de langages, mais aussi comme une machine réalisant un certain travail; c'est le cas des automates finis dits accepteurs de langages ou « reconnaisseurs » de langages.
Ce type d'automates correspond à une machine lisant au fur et à mesure une suite de symboles formant un mot pour progresser dans l'ensemble des états.
Ces automates sont munis d'un ensemble d'états appelés **états finaux** ; un mot est accepté ou reconnu par un automate si ce dernier atteint un état final.

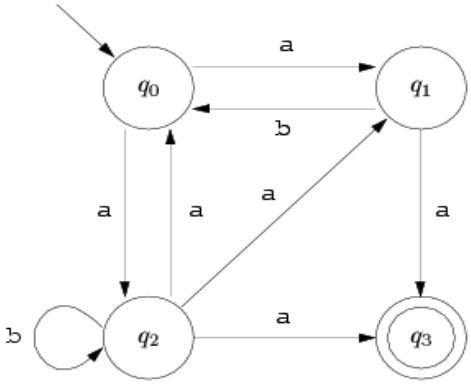
L'ensemble des mots acceptés par une telle machine forme donc un langage qui possède la caractéristique d'être régulier. On appelle langage régulier tout langage accepté par un AEF

Un automate à états finis accepteur de langage est un 5-uplet <Q, Σ, δ, q_0, F>

- Q ensemble fini des états de l'automate
- Σ l'alphabet (ensemble fini des symboles)
- δ une fonction de transition Q x Σ * → Q exprimant le passage entre deux états
- q_0 l'état initial
- F ensemble d'états finaux inclus dans Q

Exemple d'AEF accepteur de langage :





Extensions des AEF

Un automate avec transitions vides est un automate défini sur un alphabet comportant le symbole vide ε.

On appelle ε-fermeture d'un état q appartenant à l'ensemble des états Q et notée *ε-fermeture(q)* l'ensemble des états atteignables suite à des transitions vides.

2- AEF déterministes / Non déterministes

Un **AEF Déterministe** est défini par :

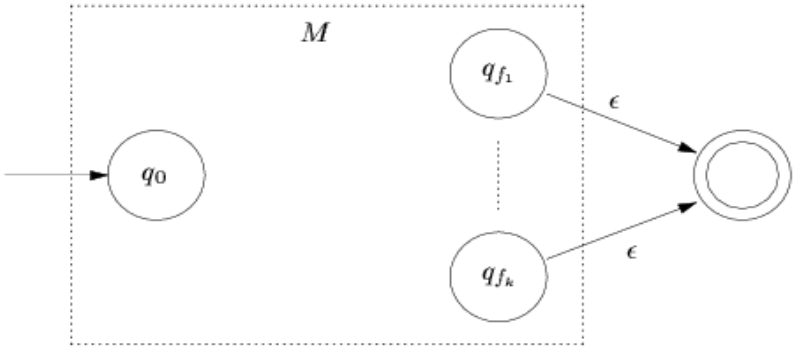
- Un seul état initial
- Une seule transition pour un même symbole
- Aucune transition sur les mots vides (ε)

Un **AEF Non Déterministe** est défini par :

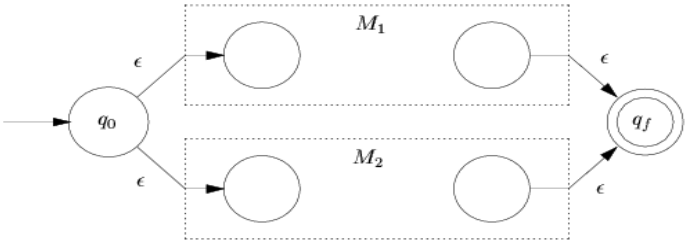
- plusieurs transitions correspondant à un même symbole dans chaque état (une fonction de transition t non plus de Q × Σ dans Q mais de Q × Σ dans P(Q), ou encore: une *relation* entre éléments de Q × Ó et de Q qui n'est pas nécessairement fonctionnelle),
- des transitions sur le mot vide (c'est à dire sans avancer sur le mot d'entrée)
- des transitions sur des mots de longueur supérieure à 1.

Equivalences et propriétés

- **Equivalence entre AEFD et AEFND**
  - **Théorème** : *pour tout langage L accepté par un AEFND, il existe aussi un AEFD qui accepte L.*
  - **Théorème (évident)** : *pour tout langage L accepté par un AEFD, il existe aussi un AEFND qui accepte L (c'est le même).*
- **Equivalence entre AEFD avec plusieurs états finaux et AEFD avec un seul état final**
  - Il suffit de rajouter des transitions vides vers un même état final



- **L'union de deux langages réguliers est un langage régulier**
- **La complémentation d'un langage régulier est un langage régulier**
- **La concaténation des langages réguliers est un langage régulier**

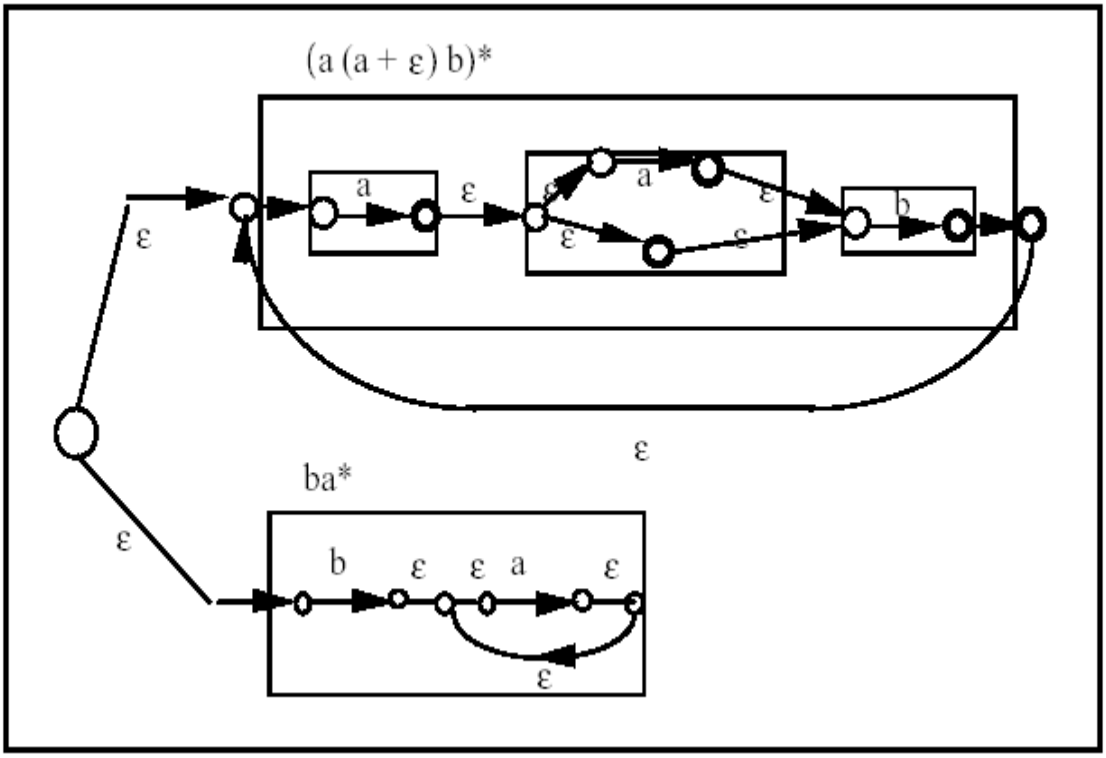


Equivalence entre AEFD et Expression Régulière

- **Equivalence entre AEFD et Expression Régulière**
  - **Théorème** : *pour toute expression régulière E, il existe un automate à états finis qui accepte le langage qu'elle dénote.*
  - **Théorème** : *tout langage accepté par un automate à états finis peut être défini par une expression régulière.*

➔ *On parle alors d'Expression régulière associée à un automate*

$(a\,(a+\varepsilon)\,b)^* + ba^*$



Algorithme de minimisation

retirer les états non accessibles; compléter l'automate

Idée : On met les états dans des classes différentes quand on voit qu'ils ont un comportement différent.

π : partition courante les états sont groupés en classes

$\pi := \{F, Q \setminus F\}$ au départ, on crée deux classes: les finaux et les non-finaux

répéter

pour chaque K, K' dans π

$\pi=\{\pi\backslash K\}U\{Ki\}$

où {Ki} est la partition de K la plus grosse

Tous les états de Ki arrivent dans la même classe pour chaque caractère

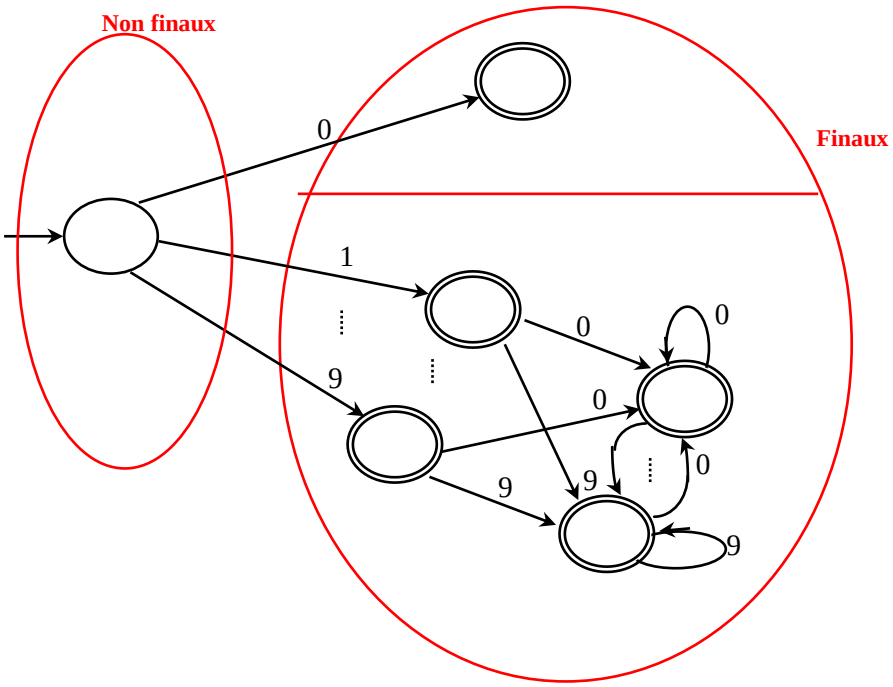
jusqu'à ce que p ne change pas

Fusionner les états d'une même classe

Retirer l'état mort

Exemple de minimisation

Minimisation du nombre d'états



π: partition des états

3- Automates à pile

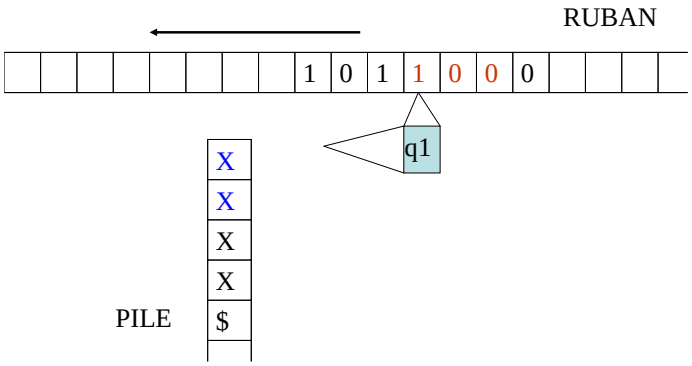
Un automate à pile est donné par :

- Q : ensemble d'états
- Σ : alphabet (ruban)
- Γ : alphabet (pile)
- $ ∈ Γ : symbole initial de pile
- q0∈Q : état initial
- F⊆ Q : ensemble d'états finaux
- Δ : ensemble de transitions (quintuplets)

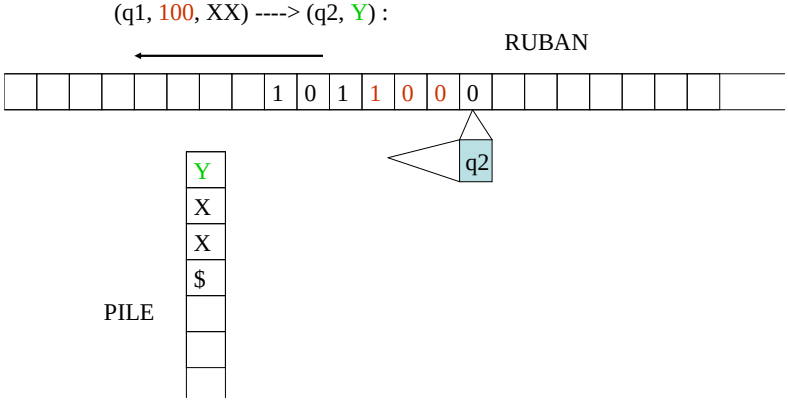
Transitions

- Un élément de Δ :
  - (q, u, α) ----> (q ',β)
    q, q '∈Q; u, v∈Σ* ; α, β ∈ Γ*
- si dans l'état q, l'automate peut lire le mot u sur le ruban (de gauche à droite) et si le mot α figure en haut de la pile (de haut en bas), alors il peut passer dans l'état q', parcourir u et remplacer en haut de la pile α par β.

(q1, 100, XX) ----> (q2, Y) :



(q1, 100, XX) ----> (q2, Y) :



Cas particuliers

- (q0, u, ε) ----> (q1, β):
  - EMPILER β
- (q0, u, α) ----> (q1, ε):
  - DEPILER α

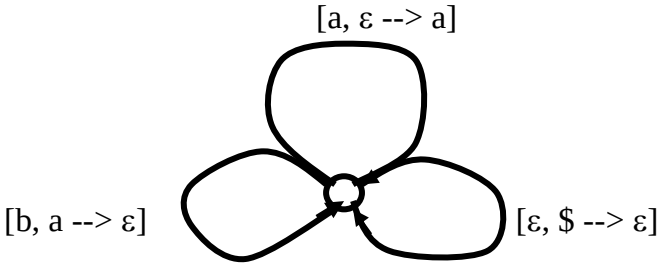
Configurations

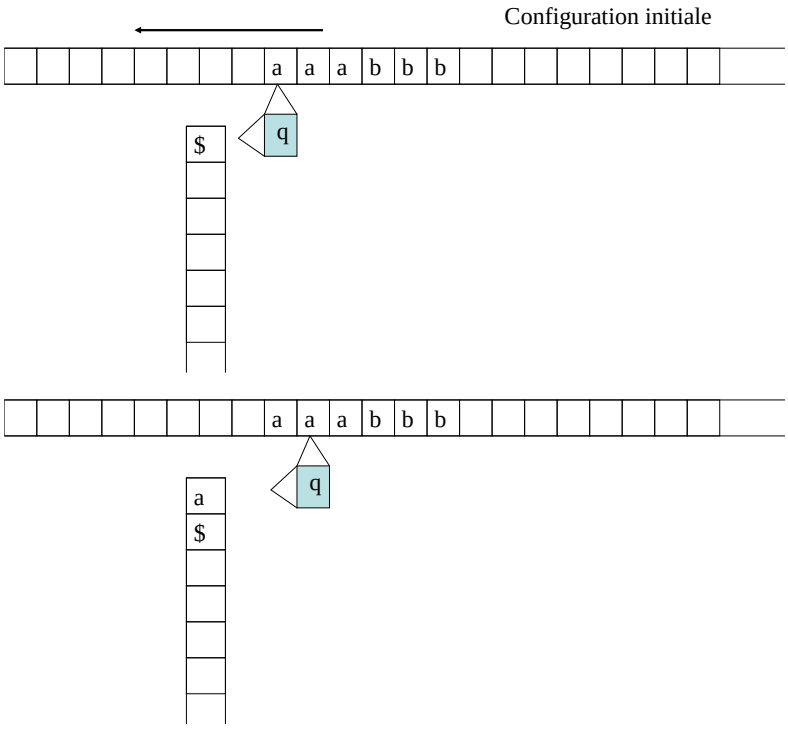
- Triplet (q, u, σ) où:
  - q∈Σ; u∈Σ*; σ∈ Γ*
  - q : état courant
  - u : mot restant à lire (de gauche à droite)
  - σ : contenu de la pile (de haut en bas)
- **configuration initiale**:
  - (q0, w, $)
- **configuration terminale**:
  - (q, ε, β) où q∈F : automate acceptant sur état final
  - (q, ε, ε) : automate acceptant sur pile vide

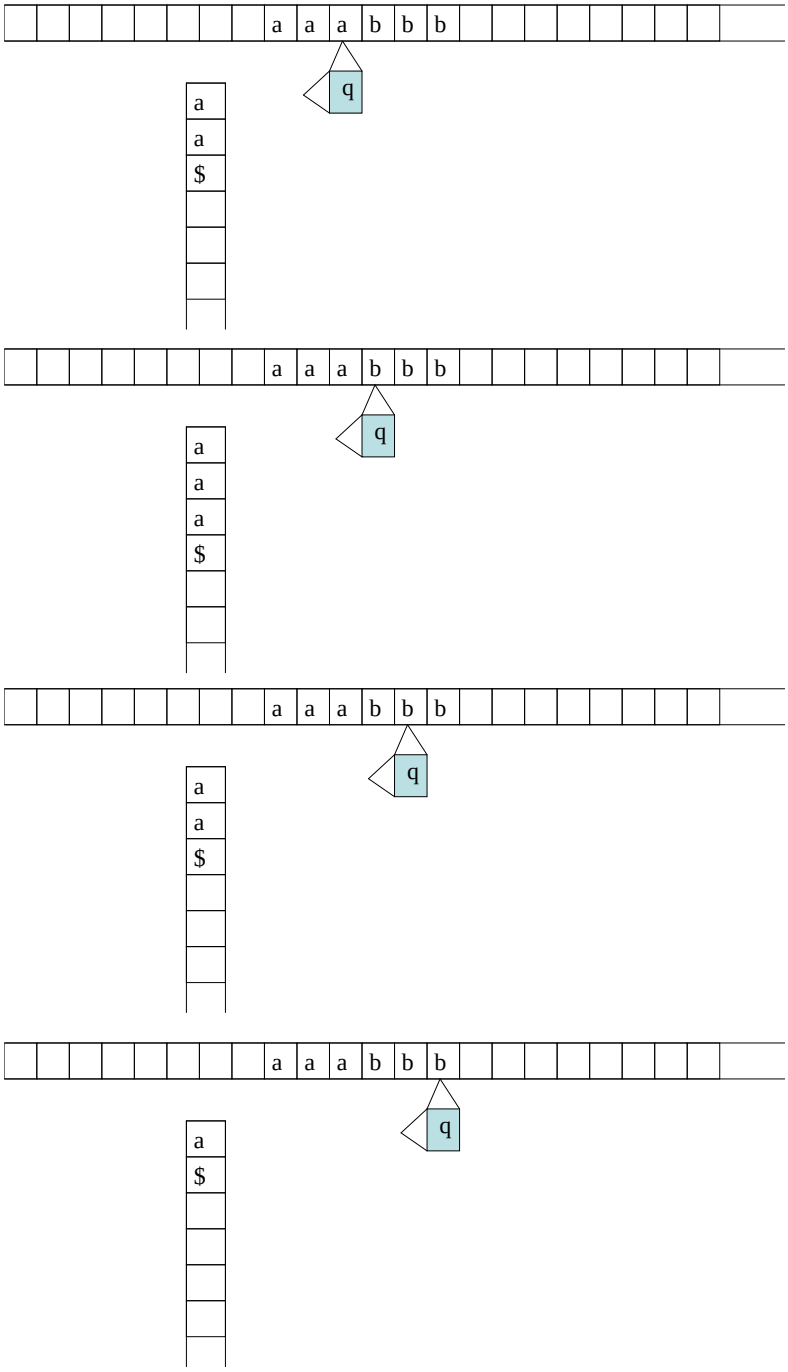
Exemple :

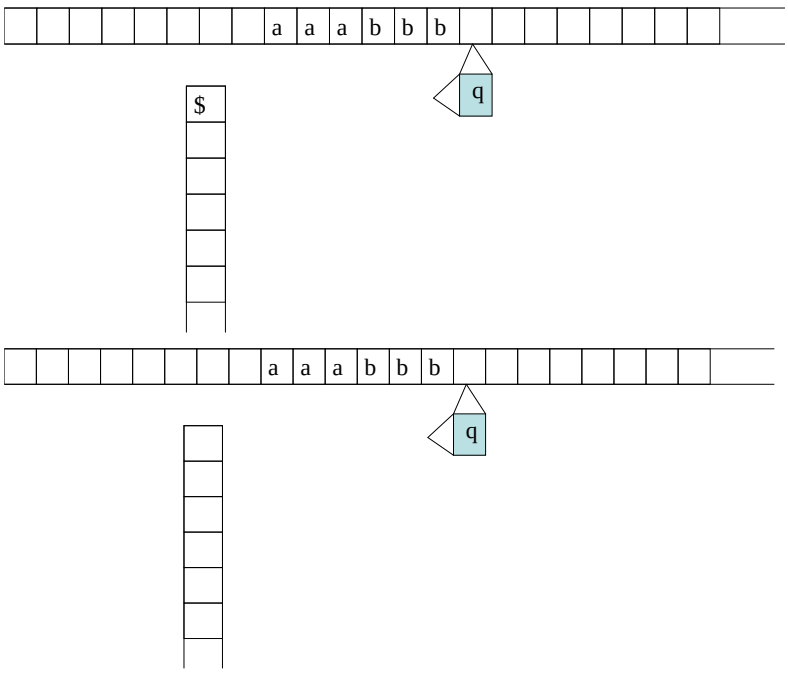
- Reconnaître $\{a^n b^n; n \geq 1\}$ sur pile vide
- **principe** : empiler des ‘ a ’ tant que l’automate lit des ‘ a ’, les dépiler quand il lit des ‘ b ’
- conséquence : un automate à pile sait « compter »

- (q, a, ε) ----> (q, a)
- (q, b, a) ----> (q, ε)
- (q, ε, $) ----> (q, ε)






a a a b b b
q
a
a
$
a a a b b b
q
a
a
a
$
a a a b b b
q
a
a
$
a a a b b b
q
a
$


a a a b b b
q
$
a a a b b b
q

Chapitre 4 ANALYSE SYNTAXIQUE

1- Grammaires hors-contextes (BNF)

Ce sont des mécanismes de description précis, facile à comprendre de la syntaxe de langages avec possibilité de construction de « reconnaisseurs » à partir de grammaires. Ils facilitent la génération de code, la détection d'erreurs etc.
En pratique les analyseurs syntaxiques nécessitent des formes restreintes de grammaires (LL, LR) qui sont :

- non ambiguës
- non récursive gauche

On peut procéder soit à une :

- Analyse descendante : (top-down) construction de l'arbre de la racine vers les feuilles
- Analyse ascendante : (bottom-up) construction de l'arbre des feuilles vers la racine

2- Analyse descendante

Pour déterminer si une *phrase* fait partie d'un langage défini par la grammaire G

- Construire un arbre de dérivation à partir de la racine
- Principe: parcours de la chaîne d'entrée de gauche à droite
- Pour chaque *token*, déterminer quelle règle appliquer
- Symbole *lookahead* - token courant

Types d'analyseurs

- *Analyseur **prédictif*** - sans retour arrière
- *Analyseur **avec retour arrière*** (backtracking)

Exemple:
type -> simple | ^id | array [simple] of type
simple -> integer | char | num dotdot num

Phrase: array [num dotdot num] of integer

Analyse descendante récursive
Elle consiste en des fonctions récursives. Une fonction est associée à chaque non terminal et le symbole lookahead détermine sans ambiguïté la fonction à appeler.

Analyse descendante récursive

Exemple: pseudo-code d'analyseur pour la grammaire précédente

```
fonction matcher(token t)
  si (lookahead = t)   alors   lookahead = prochain_token()
  sinon     erreur()
finfonction matcher

fonction type()
```

```
 si ( (lookahead == integer) ||      (lookahead == char) ||      (lookahead == num) )     alors
simple()
 sinon
  si (lookahead == '^')   alors
    matcher('^')
    matcher(id)
  sinon
   si (lookahead == array) alors
     matcher(array)
     matcher('[')
     simple()
     matcher(']')
     matcher(of)
     type()
```

Analyse descendante récursive

```
sinon
     erreur()
   finsi
  finsi
 finsi
finfonction

fonction simple()
 si (lookahead == integer)  alors    matcher(integer)
 sinon
  si (lookahead == char)  alors     matcher(char)
  sinon
   si (lookahead == num)
     matcher(num)
          matcher(dotdot)
     matcher(num)
   sinon      erreur()
   finsi
  finsi
 finsi
finfonction
```

Analyse prédictive

L'Analyse prédictive suppose la connaissance du premier symbole généré par la partie droite des productions.

C'est ainsi qu'on sait que *simple* débute par *integer*, *char*, *num* dans la production *type*.

α chaîne de symboles d'une grammaire, PREMIER(α) ensemble de terminaux débutant les chaînes dérivées de α

$\varepsilon \in$ PREMIER(α) si $\alpha \ast\Rightarrow \varepsilon$

A non-terminal, SUIVANT(A) ensemble de terminaux *a* tels qu'il existe une dérivation de la forme $S\ast\Rightarrow \alpha A a \beta$

\$ $\in$ SUIVANT(A) si A peut-être le symbole le plus à droite

Exemple

E $$ T E'
E' $$ + T E' | ε
T $$ F T'
T' $$ * F T' | ε
F $$ (E) | id

PREMIER(E) = {(,id}
PREMIER(E') = {+,eps}
PREMIER(T') = {*,eps}
SUIVANT(E) = {),$}
SUIVANT(T) = {+,),$}
SUIVANT(F) = {+,*,),$}

Détermination de PREMIER

Pour tout X symbole de la grammaire

1. si X est un terminal, PREMIER(X) est {X}
2. si X→ε , ajouter ε à PREMIER(X)
3. si X est un non-terminal et X→Y1Y2…Yk est une production,
   1. ajouter *a* à PREMIER(X) s'il existe i tel que *a*∈PREMIER(Yi) et ε ∈ PREMIER(Y1),… , PREMIER(Yi-1)
   2. ajouter ε à PREMIER(X) si ε ∈ PREMIER(Yi) pour tout i=1,2,…k

Chapitre 5 TYPAGE

1- La notion de type

Un type est défini par

- un ensemble de valeurs possibles
- les opérations portant sur ces valeurs

Les types permettent de caractériser les données manipulées par le programme. Ce sont des contraintes :

- pour fixer l'interprétation des bits en mémoire
  01000001 = 65 (vu comme un entier) ou 'A' (comme caractère)
- pour s'assurer qu'opérations et valeurs aient un sens
  dépassement de capacité: 65535+1 = 0
- opération incompatible: log (true)
- pour différencier des valeurs logiquement distinctes

  Ex : pommes ≠ poires

Les types doivent être aussi proches que possibles des concepts du monde réel (le problème à résoudre) et sont un support pour l'abstraction.
Les types expriment des contraintes qui sont contrôlées par le langage (statiquement ou dynamiquement). Ils permettent ainsi d'automatiser la vérification de la cohérence du logiciel et ces contraintes au service du programmeur.
Les violations de type constituent des erreurs de nature sémantique (statique ou dynamique)

2- Erreurs sémantiques statiques/ dynamiques

PROGRAM test;
VAR
bool : boolean;
entier : integer;
petit : 0..32;
BEGIN
bool := entier; ***Erreur de typage statique***
entier := 34; OK
petit := entier; ***Erreur de typage dynamique***
END.

Avant la dernière affectation, le compilateur Pascal aura généré le code de vérification suivant:
IF (entier < 0) OR (entier > 32) **THEN**
BEGIN
writeln('Fatal: dépassement de capacité');
halt;
END;

Un nom est en général attribué à chaque type. Dans les langages évolués, toute valeur a un type mais dans les langages de bas niveau, presque tout se confond en une notion de *mot-machine* (chaîne de bits, de taille propre à la machine sous-jacente).

3- Les différents types

Certains types sont prédéfinis (dépendant du langage):

- Entier, caractère: pratiquement universels
- Nombre réel, vecteur: langages algorithmiques
- Liste: langages de l'IA (lisp, prolog)
- Fonction, objet: langages fonctionnels, à objets

On distingue :

Types simples

Les types *simples* (ou *scalaires*) comprennent les types *discrets* (ou *ordinaux*), dont les valeurs peuvent être énumérées:

- entiers { …,-1,0,1,2,…}
- booléens { vrai, faux }
- caractères { …,'?','@','A','B',... }
- intervalles { 0..limite_sup }
- énumérés { lundi, mardi, … }

Les types simples prédéfinis sont parfois dits *primitifs m*ais les types primitifs désignent parfois tous les types prédéfinis du langage considéré, y compris des types structurés tels que les chaînes de caractères…

Type *atomique*

Ce sont des valeurs indécomposables et occupant au maximum un mot-machine (entiers, booléens).

Les types *atomiques* sont un sous-ensemble des types simples

- Un type atomique est un type de donnée manipulable en une instruction machine indivisible, et dont la valeur sera toujours cohérente même si le processus courant est provisoirement interrompu.
- Typiquement: le *mot-machine* (toute entité de taille supérieure ou inférieure risque un traitement en plusieurs étapes)

Types réels

- à virgule flottante: 2.7777777778E-10
- à virgule fixe (les *rationnels* de Ada)

  Ex: type Volt **is delta** 0.125 **range** 0.0... 255.0;

Types abstraits

Un type abstrait est un type dont la définition n'est pas rendue publique: principe **d'*encapsulation.***

Le but est de cacher l'implémentation, de manière à pouvoir facilement la changer plus tard si nécessaire.

L'accès au contenu d'un type abstrait ne se fait qu'indirectement à travers des fonctions et des procédures (les *accesseurs*) et ceci diminue la dépendance du code vis-à-vis de la représentation interne, donc des détails d'implémentation.

- Exemple d'utilisation d'un type abstrait, qui fonctionnera toujours, que '**lePoint**' soit implémenté en coordonnées cartésiennes ou polaires: **translater (lePoint, dx, dy).**

Types structurés

Par opposition, un type *structuré* est une composition d'éléments de type simple.

- enregistrements / articles (*record, class* ou *struct*)
  Ex: TYPE z = **RECORD** re, im : real **END**;
- enregistrements à variantes (*union, record.. case*)
- vecteurs (*array*)
- ensembles (*set*)
- listes ([*tete*|*queue]* en prolog, *car* et *cdr* en lisp)
- fichiers (*file*)

Les mots-clés tels que 'record', 'array' ou 'file' sont des *constructeurs de types* qui engendrent de nouveaux types.

Exemple d'enregistrement à variantes

```
type Figure (Genre: (Triangle, Carre) := Carre) is
record
 couleur : TypeCouleur := Rouge ;
 case Genre of
  when Triangle =>
   pt1,pt2,pt3 : Point;
  when Carre =>
   supgauche : Point;
   longueur : INTEGER;
 end case;
end record;
```

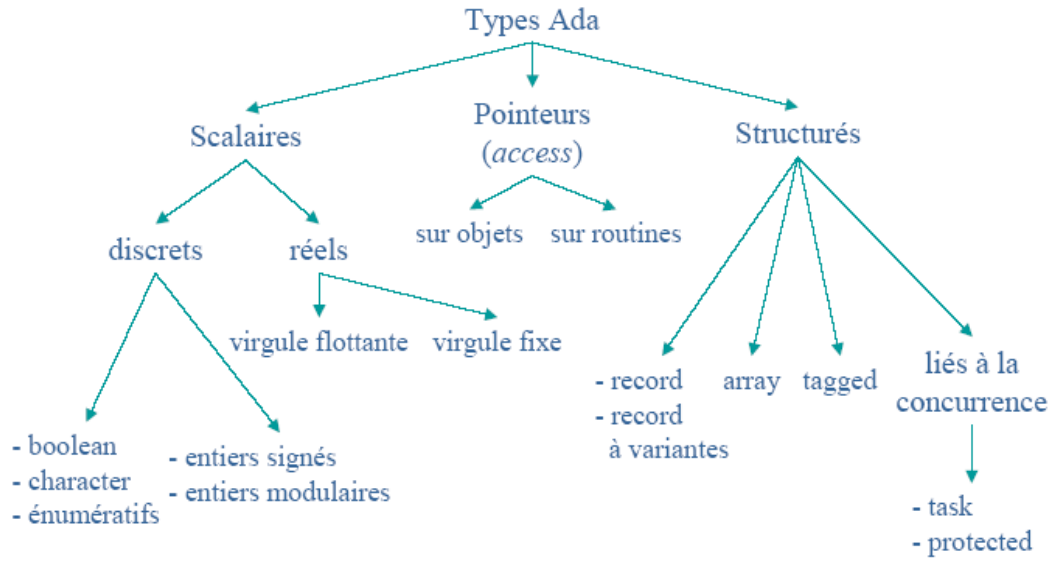
La même zone mémoire sert à stocker les différentes alternatives: superposition des variantes (mutuellement exclusives), d'où gain de place
Le *discriminant* (ici: Genre) est considéré comme un champ supplémentaire, commun à toutes les variantes.

En Ada, une instruction **proc(fig.longueur)** sera complétée par le compilateur avec un test tel que:

```
if fig.Genre = Carre then
proc(fig.longueur); -- tout est OK
else
raise constraint_error; -- signaler l'erreur
end if;
```

En Pascal et C (mais pas Ada): conversions brutes possibles entre alternatives (en l'absence de discriminant), d'où risque d'erreurs sémantiques et même de corruption de la mémoire.
La programmation par objets a rendu ce type d'approche assez désuète (sauf en programmation système).

Hiérarchie de types du langage Ada95



4- Typage faible, fort

Un langage est dit à ***typage faible*** s'il n'effectue pas (ou peu) de vérifications comme le langage C.

int i; float f; char c;

i = f*c;

Un langage est dit à ***typage fort*** s'il exige que les vérifications soient faites systématiquement comme Java, Ada, Pascal.

5-Typage statique, dynamique

Un langage est dit à ***typage statique*** s'il est fortement typé et s'il effectue (l'essentiel de) ses tests à la compilation.

A l'extrême, la notion de type peut donc être une notion qui n'existe que dans le compilateur, et qui ne laisse pas de trace dans l'implémentation.

C'est l'exemple d'Ada bien que beaucoup de tests se fassent aussi à l'exécution.

Un langage à ***typage dynamique*** effectue les contrôles (uniquement ou essentiellement) à l'exécution comme :

- Lisp, Prolog, Smalltalk
- Tous les langages à portée dynamique

6-Contrôle du typage

Pour effectuer les contrôles de typage, les langages disposent de règles: équivalence de type, compatibilité de type et inférence de type.

L'*équivalence de type*

Elle définit si deux types sont considérés identiques. On peut avoir :

- ***équivalence structurelle***: on se base sur la représentation interne (Modula-3, ML)
- ***équivalence de nom***: c'est le nom du type qui est déterminant (Ada, Java).

Cette approche est préférée actuellement, car elle supporte l'abstraction : deux types logiquement distincts sont considérés différents même si leur représentation interne est identique. Ex: coordonnées cartésiennes et polaires

Compatibilité de type

La ***compatibilité de type*** détermine quand un objet d'un certain type peut être utilisé dans un certain contexte.

S'il y a équivalence entre le type de l'objet et le type attendu par le contexte, pas de problème sinon, l'usage est légal si :

- il est possible d'effectuer une ***coercion*** (ou ***conversion implicite***) de type

  Exemple en Pascal:

  ```
  i : integer; j : 0..10;
  i := j;
  ```

- il y a un ***forçage*** (ou ***conversion explicite***) de type

  Exemple Ada avec conversion et test à l'exécution:

  ```
  n : integer; r : real;
  n := integer(r); -- overflow ?
  ```

Inférence de type

L'***inférence de type*** consiste pour le compilateur à déterminer, à partir du type des sous-expressions, le type résultant pour l'ensemble de l'expression analysée.

Elle est difficile en présence d'opérateurs génériques ou surchargés, et de conversions implicites permises.

Exemple en C++: pour la 3ème instruction, le compilateur doit déterminer qu'il y a coercion de *i* en *float* (nombre réel), puis reconversion du résultat de la multiplication en *int* (nombre entier).

```
int i = 8; float k = 3;
int r = i*k;
```

7- Sous-typage

Dans certains langages, le programmeur peut définir des *sous-types* d'un type défini au préalable par lui-même ou par le langage.

Si *S* est un *sous-type* de *T*, alors une expression de type *S* peut utilisée partout où il est légal d'en utiliser une de type *T*; une conversion de type implicite prend alors place.

Le sous-typage est largement employé dans les langages orientés objets

Un sous-type est souvent un type dont le domaine de valeurs est contraint par rapport à celui du super-type.

- Exemple Ada:

```
TYPE Jours IS (lu, ma, me, je, ve, sa, di);
SUBTYPE Weekend IS Jours RANGE sa..di;
J : Jours;
W : Weekend;
…
J := W; -- pas de problème
W := J; -- exception si J n'est ni 'sa' ni 'di'
```

8- Identificateurs dans les langages informatiques

Le nommage est aussi un support pour construire l'abstraction de manière incrémentale.

Les règles varient d'un langage à l'autre (syntaxe). Il y a la possibilité d'alias ou d'homonymes (avec durée de vie, visibilité**).**

Un nom est un moyen de se référer aux différentes entités d'un programme. On désigne quelque chose par un nom si on veut créer cette chose, l'utiliser, la changer ou la détruire.
Les entités qui ont besoin d'un nom sont:

- Les constantes et les variables, les opérateurs,
- Les labels, les types,
- Les procédures et les fonctions
- Les modules et les programmes,
- Les fichiers et les disques,
- Les commandes et les items de menus,
- Les ordinateurs, les réseaux et les usagers
-

Liaison (en anglais: *binding*)

La ***liaison*** est l'action d'associer une signification à un symbole; elle a lieu:

- à la définition du langage (signification des mots-clés)
- à la création du compilateur et des librairies de support
  - dépendances par rapport au système et au matériel
- à l'écriture et la compilation de l'application
- à l'édition de liens
- au chargement
- à l'exécution

Plus les liaisons s'établissent tôt, plus l'application s'exécutera efficacement, mais ce sera au détriment de la flexibilité.

Variables

Une variable est une abstraction de cellule de mémoire ou d'une collection de cellules.
Une variable peut être caractérisée par un 6-uplet: (Nom, adresse, valeur, type, durée de vie, portée)
L'usager décide du nom et du type de la variable. L'emplacement de la déclaration décide de sa portée et de sa longévité. Son adresse est déterminée pendant l'exécution et sa valeur dépend des instructions dans lesquelles elle apparaît.

Exemples de liaisons de variables

- Variable →nom
  - Compile time – déclarations
- Variable → adresse
  - Load time ou run time (ex : Pascal) – phénomène implicite
- Variable -> type
  - Compile time (ex : Pascal); run time (ex. Smalltalk) – déclarations
- Variable →valeur
  - Run time ou load time (initialization) – Instructions, surtout affectation.
- Variable → durée de vie
  - Compile time – Déclarations
- Variable → portée
  - Compile time – Placement des déclarations.

Portée, visibilité, durée de vie

La ***portée*** d'un identificateur correspond aux parties du programme où cet identificateur a une liaison (donc où il réfère au même 'objet')
La ***visibilité*** d'un identificateur correspond aux endroits où cet identificateur peut être utilisé.

Un identificateur peut être invisible à l'intérieur de sa portée, s'il est temporairement caché par un homonyme
La ***durée de vie*** d'un objet en mémoire (en particulier d'une variable) correspond à la période allant de sa création (allocation de mémoire) à sa destruction (libération de cette mémoire), en passant par sa (ses) portée(s).

- Plusieurs portées en cas d'alias
- Portée > durée de vie = erreur de programmation !

L'allocation de mémoire pour un objet se fait ou bien au chargement (load-time) ou à l'exécution (run-time).
Il existe donc deux classes de variables:

- Les variables statiques: l'allocation est faite une seule fois, avant que le programme commence son exécution
  - variables *globales*
  - variables locales *rémanentes* (*static* en C)
- Les variables dynamiques: l'allocation est faite pendant l'exécution du programme
  - allocation explicite: dans le tas (avec *new*, *malloc*)
  - allocation implicite: sur la pile (variables locales *automatiques*)

Blocs

Le concept de *bloc* est un des fondements de la programmation structurée**.** Les déclarations et instructions sont groupées en blocs afin de:

- Grouper les étapes d'une instruction non élémentaire
- Interpréter les noms adéquatement.

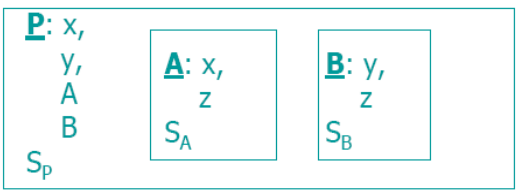
Les blocs peuvent parfois être emboîtés. Les noms introduits dans un bloc s'appellent les liaisons locales. Un nom mentionné mais pas défini dans un bloc doit avoir été défini dans l'un des blocs de l'englobant.

Blocs anonymes et blocs nommés

Un programme, une procédure ou une fonction sont des exemples de **blocs nommés**.
Un **bloc anonyme** est comme une procédure sans nom qui est appelée immédiatement lors de son occurrence dans le texte du programme.
Les blocs anonymes sont utiles lorsqu'un calcul est nécessaire une seule fois et que ses variables ne sont utiles que dans ce calcul: on ne veut pas les déclarer à l'extérieur de ce bloc.
Les blocs anonymes existent p.ex. en Ada et en C, mais en C, les blocs nommés imbriqués ne sont pas permis.

Visibilité, masquage d'identificateurs

Si le même nom X est défini dans un bloc environnant B1 et dans un bloc emboîté B2 (d'où B2 $\subset$ B1), alors la visibilité du X défini en B1 est perdue dans le bloc B2 qui ne voit que le X défini en B2.



Portée statique et dynamique

Dans les langages à ***portée statique*** (ou *lexicale*), la portée des identificateurs est fixée à la compilation de manière à être le plus petit bloc englobant leur déclaration.
La ***portée dynamique*** cherche l'identificateur dans la chaîne des procédures appelantes. Cette chaîne considère les règles de visibilité mais pas l'emboîtement.
La portée dynamique a surtout l'avantage d'être plus simple à implémenter: pas de notion de *lien statique* (pointeur sur le bloc d'activation du bloc englobant) en plus du ***lien dynamique*** (pointeur sur le bloc d'activation de l'appelant).

Exercice

- Quel résultat nous donnerait:
  - La portée statique
  - La portée dynamique

```
PROGRAM P;
VAR X: integer;
PROCEDURE A;
BEGIN
X:= X+1;
print(X);
END;
PROCEDURE B;
VAR X:integer;
BEGIN
X:= 17;
A;
END;
BEGIN (* P *)
X:= 23;
B;
END;
```

Surcharge

La ***surcharge*** (ou ***polymorphisme ad-hoc***; ***overloading*** en anglais) consiste à donner plusieurs significations concurrentes à un même symbole (en général identificateur de fonction ou procédure).
Les opérateurs mathématiques sont surchargés par défaut dans la plupart des langages

- le "-" peut être *monadique* ou *dyadique*
- le "+" opère sur les entiers et les réels

Ada et C++ permettent au programmeur de définir des opérations surchargées;
Le compilateur exigera de pouvoir les distinguer par leur *profil* (nombre et type des arguments, y compris un éventuel paramètre résultat).
Elle est pratique pour signifier intuitivement une sémantique similaire à d'autres identificateurs identiques connus.
La surcharge est un ***sucre syntaxique*** : ce n'est qu'un raccourci pour le confort du programmeur. Mais l'abus peut avoir un effet inverse et un impact négatif sur l'interopérabilité: la surcharge rend difficile l'exportation de définitions vers des programmes écrits dans d'autres langages.

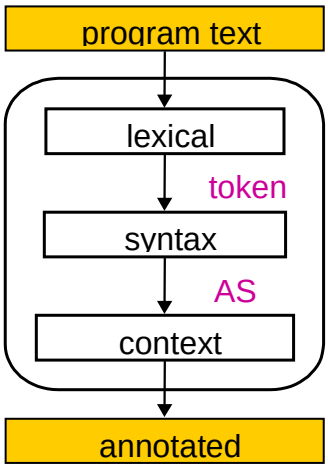
Chapitre 6 ANALYSE SEMANTIQUE

1-Introduction

On a besoin de structures de données pour représenter le programme à compiler comme :

- Table des symboles
- Arbre syntaxique décoré (Annotated Syntaxic Tree)

L'implémentation d'un compilateur n'a pas besoin de sérialiser les différentes phases d'où une construction incrémentale de l'AST.



L'information est éparpillée tout le long du programme source et les identificateurs servent de connecteurs d'où il faut garder une trace de la définition des identificateurs. Ainsi si on a :

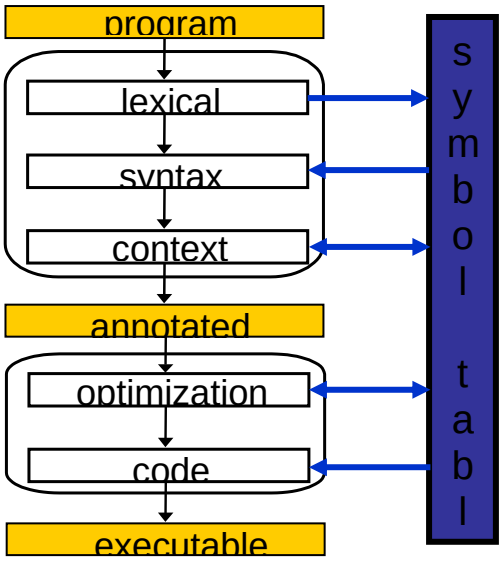
- Des identificateurs non définis ⇒ error (erreur)
- Des identificateurs non utilisés ⇒ warning (avertissement)

On construit un arbre syntaxique des tokens rencontrés et le « décore » avec des informations sémantiques (valeur d'un identificateur etc.)
L'arbre syntaxique (AST) définit l'ordre d'exécution des instructions et le flux de données.
On procède également à la vérification des règles de définition du langage à savoir le typage et le contrôle de flux (dead code).

2- Table des Symboles

Elle est utilisée par toutes les phases de compilation et garde des informations sur les identificateurs à savoir :

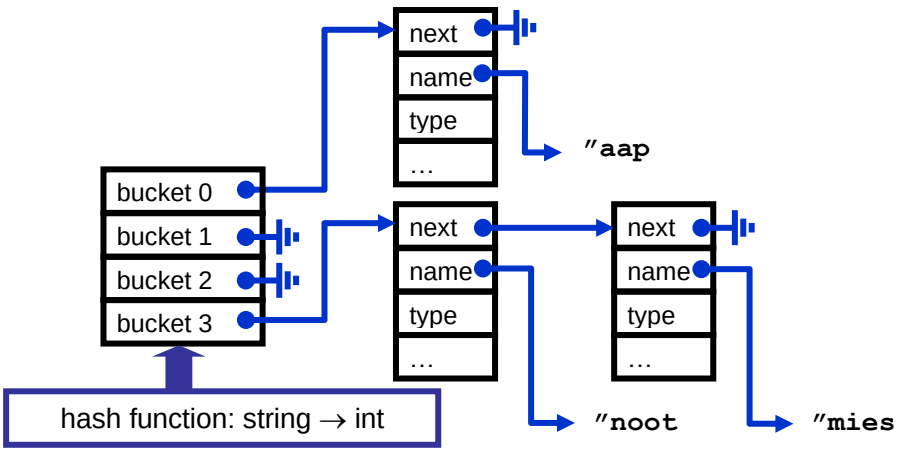
- type
- position
- taille
- Valeur



Implémentation de la table des symboles

C'est un tableau extensible de chaînes indexées:

- linear list
- tree
- hash table



Hachage

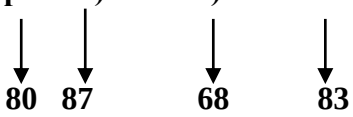
Il se base sur une fonction de hachage

- Fonction non bijective retournant généralement un entier et vérifiant :
  - x=y ➔ $F_{hash}(x)=F_{hash}(y)$
  - Il est rare mais possible que
    - x!=y et $F_{hash}(x)=F_{hash}(y)$
  - F_{hash} n'est pas une fonction uniforme c'est à dire
    - Distance(x,y) << Distance($F_{hash}(x)$, $F_{hash}(y)$)

La définition de la distance peut changer selon le contexte. Elle pourrait être par exemple le nombre de bits différents etc.

Exemple de fonction de hachage

- chaîne="pwds"
- **F_{hash}(chaîne)=((31*'p'+'w')*31+'d')*31+'s'**



- **F_{hash}(chaîne)=2469078**

C'est une bonne fonction de hachage pour les longues chaines. Elle est utilisée par Unix pour le hachage des chemins de fichiers (correspondance en chemin et descripteur de fichiers).

Table de hachage

C'est un tableau à dimension variable de listes chaînées. Chaque liste contient toutes les entrées ayant le code de hachage (hashcode).Chaque entrée contient un triplet (clef, valeur, suivant) :

- clef : la clef unique
- valeur : les informations relatives à la clef (les décorations comme la position du token, son type explicite etc.)
- suivant : un pointeur vers l'entrée suivante de la liste chaînée.

Opérations sur la table de hachage

- Insertion dans la table de la clef c
  - h=F_{hash}(c) MOD taille (tableau)
  - Ajouter une nouvelle entrée en fin de file à la position h
- Recherche dans la table de la clef c
  - h=F_{hash}(c) MOD taille (tableau)
  - Parcourir la file à la position h
  - Plus la taille (tableau) est importante plus la liste est de longueur réduite plus la recherche est rapide
- Mise à jour dans la table de la clef c
  - Recherche de l'entrée correspondant à la clef c
  - Modifier la valeur de l'entrée
- Suppression dans la table de la clef c
  - Recherche de l'entrée correspondant à la clef c
  - Modifier les liens (de la liste chaînée) pour retirer l'entrée

Identification

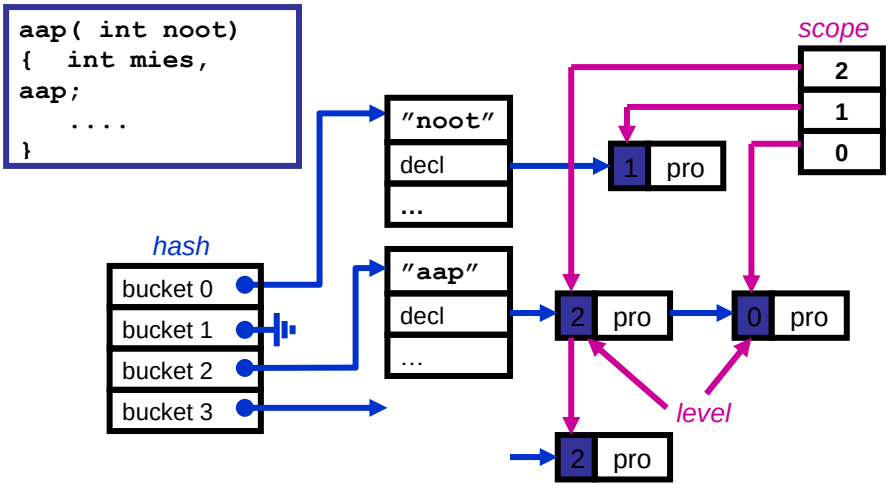
- différents types d'identificateurs :
  - variables
  - Nom de types
- name spaces
- Portée

```
typedef int i;

int j;

void foo(int j)
{
    struct i {i i;} i;

i:  i.i = 3;
    printf( "%d\n", i.i);
}
```

Gestion de la portée

- Pile de la portée des éléments
  - Entrée d'une nouvelle portée ➔ un nouvel élément est empilé
  - Les déclarations des identificateurs sont au niveau de l'élément (empilé) de portée
  - Les identificateurs rencontrés sont à consulter dans les portées de la portée la plus proche à la plus globale (de haut en bas)
  - La tête de la pile est dépilée une fois qu'on est sorti de la portée
  -

Table de symboles à base de table de hachage des portées



3- Difficultés

Au cours de l'identification les complications qui peuvent subvenir sont:

- Surcharge (Overloading) des :
  - opérateurs
  - fonctions: `PUT(s:STRING)` `PUT(i:INTEGER)`

Solution: reconnaître un ensemble de possibilités (à réduire par vérification des types)

- Import de portées
  - C++ scope resolution operator `x::`
  - Modula `FROM module IMPORT ...`

Solution: empiler ou fusionner les deux portées

Vérification de type

Les opérateurs et fonctions imposent des restrictions sur le type des arguments.
Les informations sur le type doivent être sauvegardées dans une table de type et les informations sur le type doivent être résolues (type non définis)

- Circularités

On vérifie également l'équivalence des noms (tous les types ont un nom unique)
Ex :

```
VAR a : ARRAY [Integer 1..10] OF Real;
VAR b : ARRAY [Integer 1..10] OF Real;
```

On vérifie l'équivalence des structures mais elle est difficile à réaliser.
Ex :

```
TYPE c = RECORD i : Integer; p : POINTER TO c; END RECORD;
TYPE d = RECORD
                    i : Integer;
                    p : POINTER TO
                            RECORD
                              i : Integer;
                              p : POINTER to c;
                            END RECORD;
                END RECORD;
```

```
VAR a: Real;
...
a := 5;
```

```
3.14 + 7
 8 + 9
```

Les contraintes liées à la vérification sont :

- Conversion implicites pour suivre le type des arguments
- Problème d'ambiguïté

Solution : Approche à deux phases

- étendre le type à un ensemble de types possibles
- Réduire cet ensemble en appliquant les contraintes imposées par les opérateurs ou la sémantique du langage