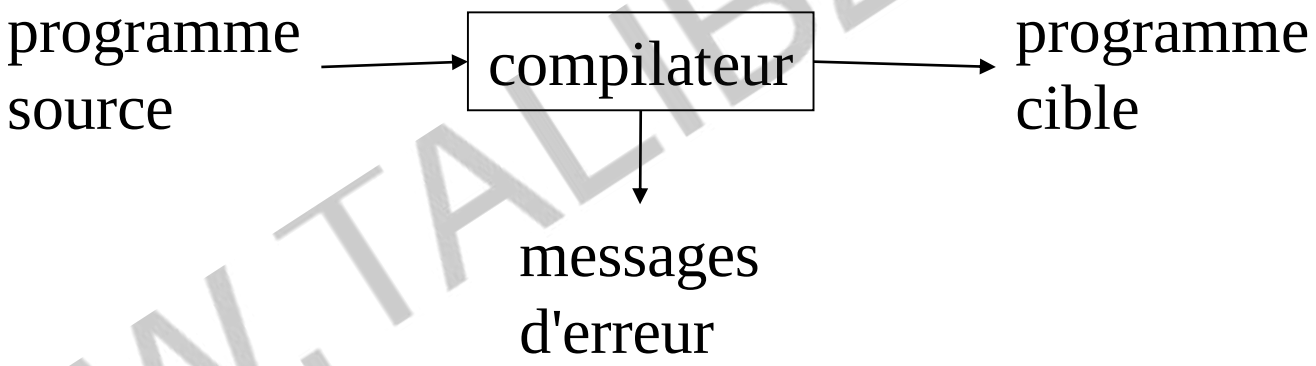
COMPILATION
ET
THEORIE DES LANGAGES

Plan

1. Introduction
2. Structure d'un compilateur
3. Analyse lexicale
4. Théorie des langages et les automates
5. L'outil (f)lex
6. Analyse syntaxique

Introduction

Un compilateur est un logiciel particulier qui traduit un programme écrit dans un langage de haut niveau (par le programmeur) en instructions exécutables (par un ordinateur). C'est donc l'instrument fondamentale à la base de tout réalisation informatique.



- Premier compilateur : compilateur Fortran de J. Backus (1957)
- Langage source : langage de haut niveau (C, C++, Java, Pascal, Fortran...)
- Langage cible : langage de bas niveau (assembleur, langage machine)

Introduction

- La compilation est aussi la traduction d'un langage de programmation de haut niveau vers un autre langage de programmation de haut niveau. Par exemple une traduction de Pascal vers C, ou de Java vers C++, on parle plutôt de **traducteur**
- Autre différence entre traducteur et compilateur : dans un traducteur, il n'y a pas de perte d'informations (on garde les commentaires, par exemple), alors que dans un compilateur il y a perte d'informations.
- la traduction d'un langage de programmation de bas niveau vers un autre langage de programmation de haut niveau. Par exemple pour retrouver le code C à partir d'un code compilé (piratage, récupération de vieux logiciels, ...)
- la traduction d'un langage **quelconque** vers une autre langage quelconque (ie pas forcemment de programmation) : word vers html, pdf vers ps, ...

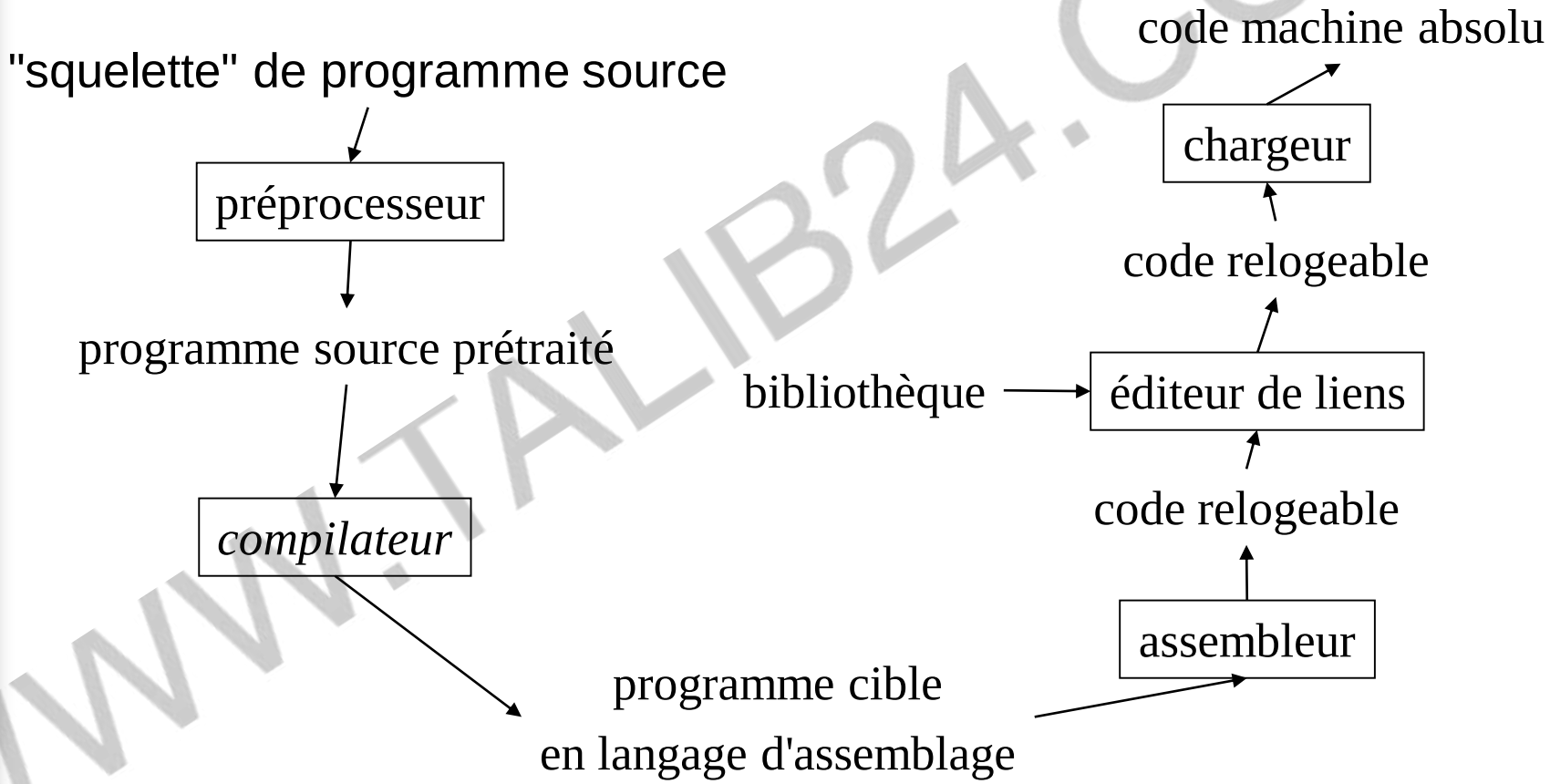
Introduction
Pourquoi ce cours ?

- Le but de ce cours est de présenter les principes de base inhérents à la réalisation de compilateurs.
- Les idées et techniques développées dans ce domaine sont si **générales et fondamentales** qu'un informaticien (et même un scientifique non informaticien) les utilisera très souvent au cours de sa carrière : traitement de données, moteurs de recherche, outils sed ou awk, etc.
- Nous verrons :
  - les principes de base inhérents à la réalisation de compilateurs : analyse lexicale, analyse syntaxique, analyse sémantique, génération de code,
  - les outils fondamentaux utilisés pour effectuer ces analyses : fondements de base de la théorie des langages (grammaires, automates, ...), méthodes algorithmiques d'analyse, ...
- En outre, comprendre comment est écrit un compilateur permet de mieux comprendre les "contraintes" imposées par les différents langages lorsque l'on écrit un programme dans un langage de haut niveau.

Bibliographie

- *Aho, Sethi, Ullman,* **Compilateurs, Principes, techniques et outils**, Interéditions 1986/1989.
- *R.Wilhelm, D. Maurer*, **Les compilateurs : théorie, construction, génération**, Masson 1994.
- *J. Levine, T. Masson, D. Brown*, **lex & yacc**, Éditions O'Reilly International Thomson 1995.

L'environnement d'un compilateur



Cousins des compilateurs
Interpréteurs

- Diffèrent d'un compilateur par l'intégration de l'exécution et de la traduction : 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.
- Utilisés pour les langages de Commande.
- L'interpréteur doit être présent sur le système à chaque fois que le programme est exécuté.

→on ne peut pas cacher le code (et donc garder des secrets de fabrication). Généralement les interpréteurs sont assez petits, mais le programme est plus lent qu'avec un langage compilé.

Exemples de langages interprétés : BASIC, scheme, CaML, Tcl, LISP, Perl, Prolog

Cousins des compilateurs
Préprocesseurs

Effectuent des substitutions de définitions, des transformations lexicales, des définitions de macros :

Traitement des macros	#define
Inclusion de fichiers	#include
Extensions de langages	#ifdef

Les définitions de macros permettent l'utilisation de paramètres.

Par exemple, en TEX, une définition de macro a la forme

\define <nom> <modèle>{<corps>}

Par exemple, si on définit :

```
\define\JACM #1;#2;#3.
  {{\sl J.ACM} {\bf #1}:#2, pp. #3}
```

et si on écrit

```
\JACM 17;4;715-728
```

on doit voir

J.ACM **17**:4, pp. 715-728

Cousins des compilateurs
Langages Intermédiaire ou P-code

- Les **langages P-code** ou **langages intermédiaires** sont à mi-chemin de l'interprétation et de la compilation.
- 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. Ce P-code est interprété pour exécuter le programme,
- Par exemple en Java, le source est compilé pour obtenir un fichier (.class) "byte code" qui sera interprété par une **machine virtuelle**. Autre langage p-code : Python.

- Les interpréteurs de p-code peuvent être relativement petits et rapides, si bien que le p-code peut s'exécuter presque aussi rapidement que du binaire compilé. En outre les langages p-code peuvent garder la flexibilité et la puissance des langages interprétés. On ne garde que les avantages !!

Cousins des compilateurs
Editeur de liens

- Une autre phase importante qui intervient après la compilation pour obtenir un exécutable est la phase d'**éditions de liens**.
- Un éditeur de liens résoud entre autres les références à des appels de routines dont le code est conservé dans des librairies.
- En général, un compilateur comprend une partie éditeur de liens. Nous n'en parlerons pas ici. En outre, sur les systèmes modernes, l'édition des liens est faite à **l'éxécution** du programme ! (le programme est plus petit et les mises à jour plus faciles).

Langages d’assemblage

- Les langages d’assemblage ou assembleurs sont des versions un peu plus lisibles du code machine avec des noms symboliques pour les opérations et pour les opérandes

```
MOV a, R1
ADD #2, R1
MOV R1, b
```

- Chaque processeur a son langage d'assemblage
- Les langages d'assemblage d'un même constructeur se ressemblent

Assemblage

- L'**assemblage** consiste à traduire ce code en code binaire
- La forme la plus simple est l'**assemblage en deux passes**

Première passe

- On crée une table des symboles (distincte de celle du compilateur) pour associer des adresses mémoires aux identificateurs

identificateur	adresse
a	0
b	4

Assemblage

Deuxième passe

- On crée du code machine relogeable c'est-à-dire relatif à l'adresse mémoire *L* où il commence
- Les instructions précédentes peuvent être traduites en :

  0001 01 00 00000000 *
  0011 01 10 00000010
  0010 01 00 00000100 *

- La table des symboles de l'asssembleur sert aussi à l'édition de liens
- Elle permet de remplacer les références à des noms externes

Langages machines

- Chaque processeur a son langage

Exemples d'instructions binaires

0001 01 00 00000000 *
0011 01 10 00000010
0010 01 00 00000100 *

- Les 4 premiers bits : **code de l'instruction :**
  0001 = charger
  0011 = ajouter
  0010 = sauvegarder
- Les 2 bits suivants : **registre**

Langages machines

- Les 2 bits suivants : **mode d'adressage**

  00 = mode adresse

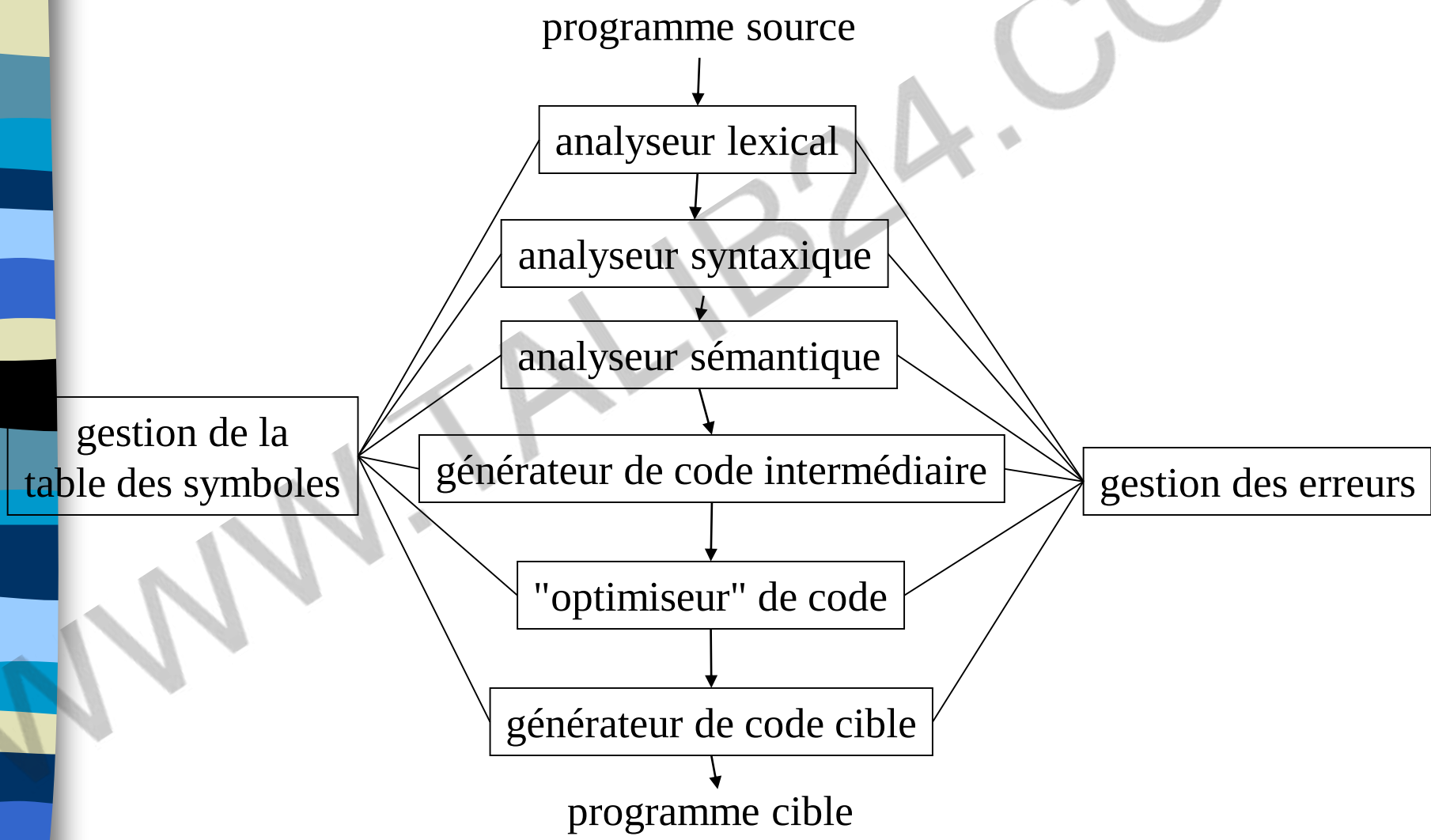
  10 = mode littéral
- L'étoile (**bit de translation**) indique que l'adresse *L* doit être ajoutée aux opérandes. Si L = 00001111, c'est-à-dire 15, a et b doivent être placés en 15 et 19
- Code absolu obtenu :

  0001 01 00 00001111

  0011 01 10 00000010

  0010 01 00 00010011

Structure d'un compilateur



Analyse lexicale

Appelée aussi *Analyse linéaire*

- Reconnaître les "types" des "mots" lus. Pour cela, on lit le programme source de gauche à droite et les caractères sont regroupés en **unités lexicales**.
- Se charger d'éliminer les caractères superflus (commentaires, espaces, ...)
- Identifier les parties du texte qui ne font pas partie à proprement parler du programme mais sont des directives pour le compilateur .
- Identifier les symboles qui représentent des identificateurs, des constantes réelles, entière, chaînes de caractères, des opérateurs (affectation, addition, ...), des séparateurs (parenthèses, points virgules, ...), les mots clefs du langage.
- Outils théoriques utilisés : expressions régulières et automates à états finis

Analyse lexicale

On passe de

position = initial + vitesse * 60

à

[id, 1] [=] [id, 2] [+] [id, 3] [*] [60]

- Les identificateurs rencontrés sont placés dans la table des symboles
- Les blancs et les commentaires sont éliminés

Analyse syntaxique

Appelée aussi *Analyse hiérarchique* ou *Analyse grammaticale*

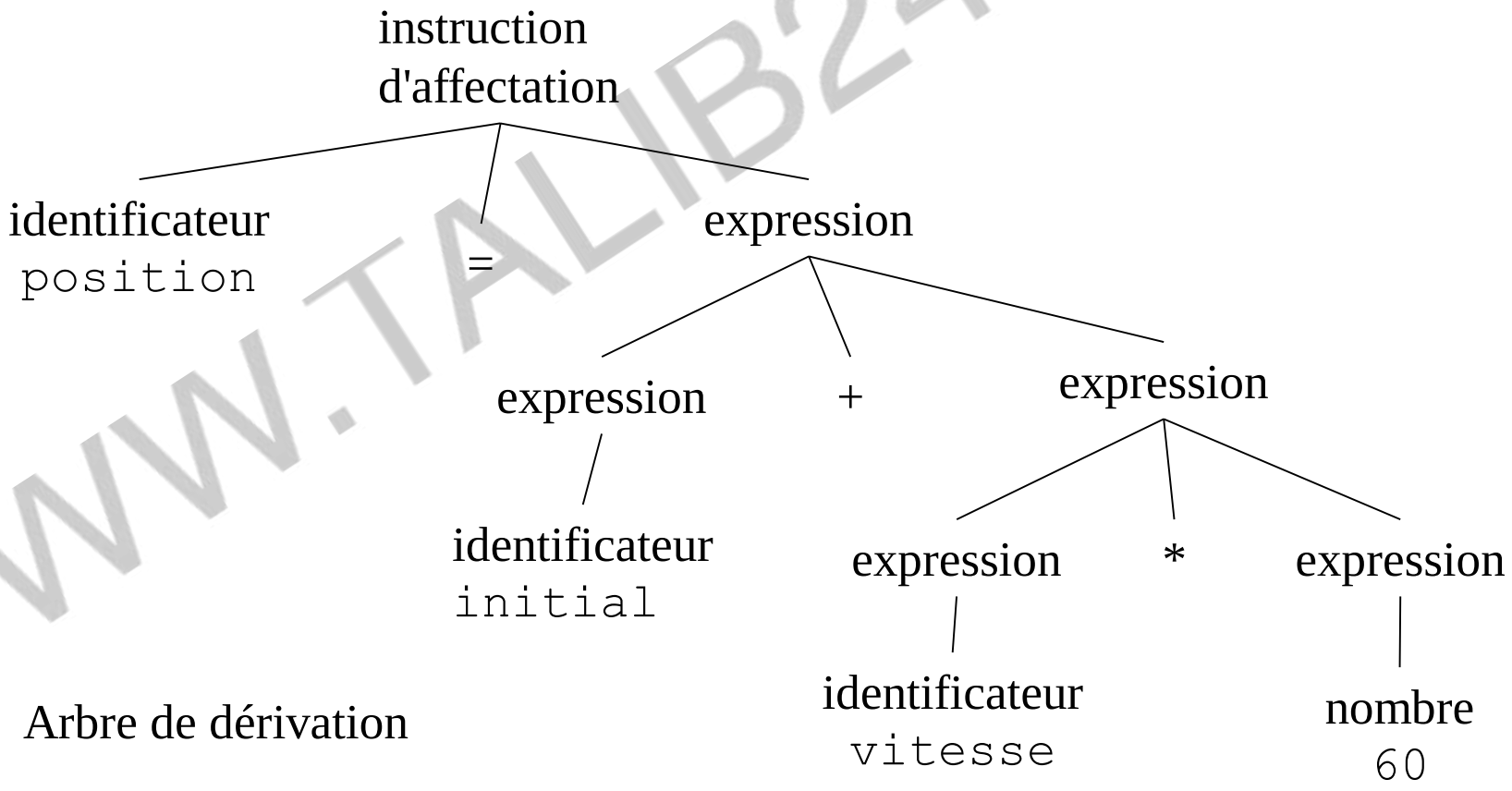
- Il s'agit de regrouper les unités lexicales en structures grammaticales, de découvrir la structure du programme. L'analyseur syntaxique sait comment doivent être construites les expressions, les instructions, les déclarations de variables, les appels de fonctions, ...

- Exemple. En C, une sélection simple doit se présenter sous la forme :
  if (*expression*) *instruction*
  Si l'analyseur syntaxique reçoit la suite d'unités lexicales
  MC_IF IDENT OPREL ENTIER ...
  il doit signaler que ce n'est pas correct car il n'y a pas de (juste après le if.

- Outils théoriques utilisés : grammaires et automates à pile

Analyse syntaxique

On reconstruit la structure syntaxique de la suite d'unités lexicales fournie par l'analyseur lexical.



Arbre de dérivation

Analyse Sémantique

Appelée aussi *analyse contextuelle*

- Dans cette phase, on opère certains contrôles (contrôles de type, par exemple) afin de vérifier que l'assemblage des constituants du programme a un sens. On ne peut pas, par exemple, additionner un réel avec une chaîne de caractères, ou affecter une variable à un nombre, ...
- Outil théorique utilisé : schéma de traduction dirigée par la syntaxe

Génération de code

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

- Mais si l'on écrit un compilateur pour un processeur donné, il n'est alors pas évident de porter ce compilateur (ce programme) sur une autre machine cible. C'est pourquoi (entre autres raisons), on introduit des machines dites **abstraites** qui font abstraction des architectures réelles existantes. Ainsi, on s'attache plus aux principes de traduction, aux concepts des langages, qu'à l'architecture des machines.

- En général, on produira dans un premier temps des instructions pour une machine abstraite (virtuelle). Puis ensuite on fera la traduction de ces instructions en des instructions directement exécutables par la machine réelle sur laquelle on veut que le compilateur s'exécute. Ainsi, le portage du compilateur sera facilité, car la traduction en code cible virtuel sera faite une fois pour toutes, indépendamment de la machine cible réelle.

Génération de code intermédiaire

- Programme pour une machine abstraite

Représentations utilisées

- Code à trois adresses

temp1 := inttoreal(60)

temp2 := id3 * temp1

temp3 := id2 + temp2

id1 := temp3

Optimisation de code

- Cette phase tente d'améliorer le code produit de telle sorte que le programme résultant soit plus rapide.
- Il y a des optimisations qui ne dépendent pas de la machine cible :
  - élimination de calculs inutiles (faits en double),
  - élimination du code d'une fonction jamais appelée, propagation des constantes, extraction des boucles des invariants de boucle, ...
- Et il y a des optimisations qui dépendent de la machine cible :
  - remplacer des instructions générales par des instructions plus efficaces et plus adaptées,
  - utilisation optimale des registres, ...

Optimisation de code

Elimination des opérations inutiles pour produire du code plus efficace.

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

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

- La constante est traduite en réel flottant à la compilation, et non à l'exécution
- Les variables temp2 et temp3 sont éliminées

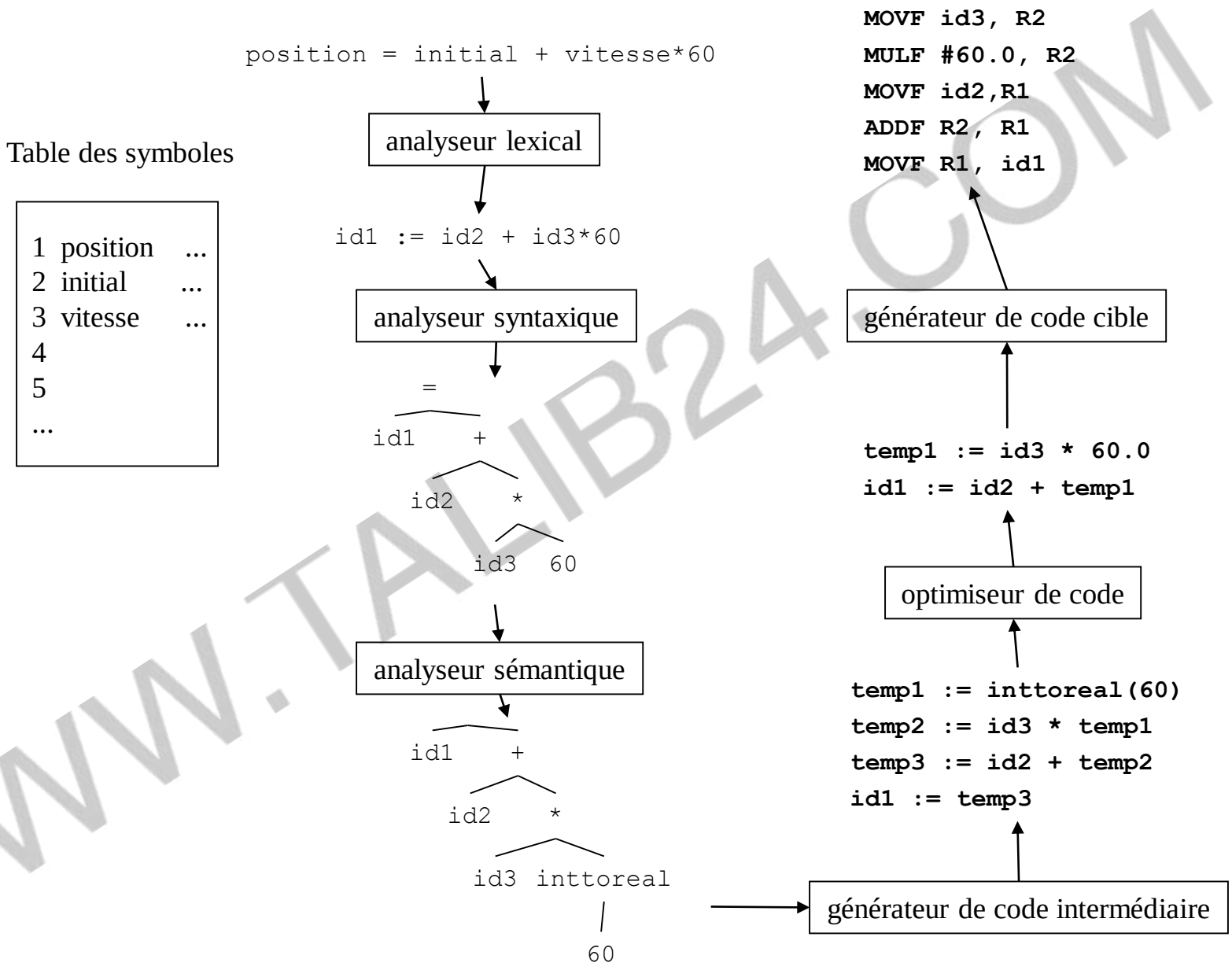
Génération de code cible

- La dernière phase produit du code en langage d'assemblage

- Un point important est l'utilisation des registres

temp1 := id3 * 60.0	**MOVF id3, R2**
id1 := id2 + temp1	**MULF #60.0, R2**
	MOVF id2,R1
	ADDF R2, R1
	MOVF R1, id1

- F = flottant.
- La première instruction transfère le contenu de id3 dans le registre R2
- La seconde multiplie le contenu du registre R2 par la constante 60.0



Gestion de la table des symboles

- La **table des symboles** est la structure de données utilisée servant à stocker les informations qui concernent les identificateurs du programme source (par exemple leur type, leur emplacement mémoire, leur portée, visibilité, nombre et type et mode de passage des paramètres d'une fonction, ...)

- Le remplissage de cette table (la collecte des informations) a lieu lors des phases d'analyse.
- Les informations contenues dans la table des symboles sont nécessaires lors des analyses syntaxique et sémantique, ainsi que lors de la génération de code.

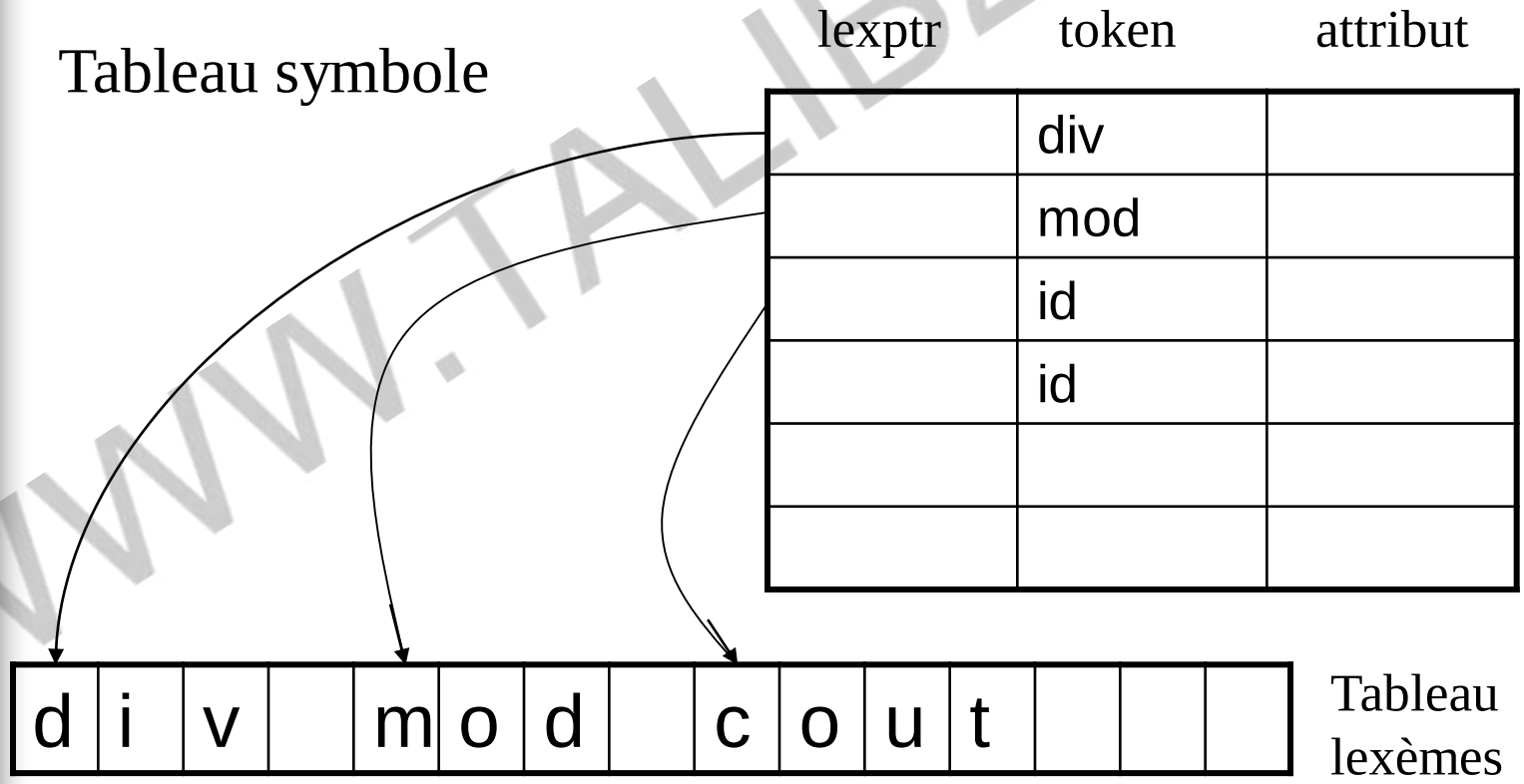
Ajout d'une table des symboles

La table des symboles utilise deux fonctions

insert(s, t) crée et renvoie une entrée pour la chaîne s et le token t

lookup(s) renvoie l'indice de la chaîne s, ou 0 si elle n'y est pas

On peut ainsi traiter les mots-clés : insert("div", div)



Gestion des erreurs

- Chaque phase peut rencontrer des erreurs. Il s'agit de les détecter et d'informer l'utilisateur le plus précisément possible : erreur de syntaxe, erreur de sémantique, erreur système, erreur interne.
- Un compilateur qui se contente d'afficher syntax error n'apporte pas beaucoup d'aide lors de la mise au point.
- Après avoir détecté une erreur, il s'agit ensuite de la traiter de telle manière que la compilation puisse continuer et que d'autres erreurs puissent être détectées.
- Un compilateur qui s'arrête à la première erreur n'est pas non plus très performant. Bien sûr, il y a des limites à ne pas dépasser et certaines erreurs (ou un trop grand nombre d'erreurs) peuvent entrainer l'arrêt de l'exécution du compilateur.

Groupement des phases

- **Partie frontale** (*front end*)
  Regroupe tout ce qui dépend du langage source plutôt que de la machine cible. On peut utiliser la même partie frontale sur une machine différente
- **Partie arrière** (*back end*)
  Regroupe le reste
- **Passes**
  Plusieurs phases peuvent être regroupées dans une même passe consistant à lire un fichier et en écrire un autre
- Analyse lexicale, syntaxique, sémantique et génération de code intermédiaire peuvent être regroupées en une seule passe
- La réduction du nombre de passes accélère le traitement

Outils théoriques

Différentes phases de la compilation		Outils théoriques utilisés
Phases d'analyse	analyse lexicale (**scanner**)	expressions régulières automates à états finis
	analyse syntaxique (**parser**)	grammaires automates à pile
	analyse sémantique	traduction dirigée par la syntaxe
Phases de production	génération de code	traduction dirigée par la syntaxe
	optimisation de code	
Gestions parallèles	table des symboles	
	traitement des erreurs	

Outils logiciels

- **Générateurs d'analyseurs lexicaux**
  Engendrent un analyseur lexical (*scanner, lexer*) sous forme d'automate fini à partir d'une spécification sous forme d'expressions rationnelles : Flex, Lex

- **Générateurs d'analyseurs syntaxiques**
  Engendrent un analyseur syntaxique (*parser*) à partir d'une grammaire : Bison, Yacc

- **Générateurs de traducteurs**
  Engendrent un traducteur à partir d'un schéma de traduction (grammaire + règles sémantiques) : Bison, Yacc

Conclusion

- Pendant toutes les années 50, les compilateurs furent tenus pour des programmes difficiles à écrire. Par exemple, la réalisation du premier compilateur Fortran nécessita 18 hommes-années de travail (1957).
- On a découvert depuis des techniques systématiques pour traiter la plupart des tâches importantes qui sont effectuées lors de la compilation.

Conclusion

- Contrairement à une idée souvent répandue, la plupart des compilateurs sont réalisés (écrits) dans un langage de haut niveau, et non en assembleur.

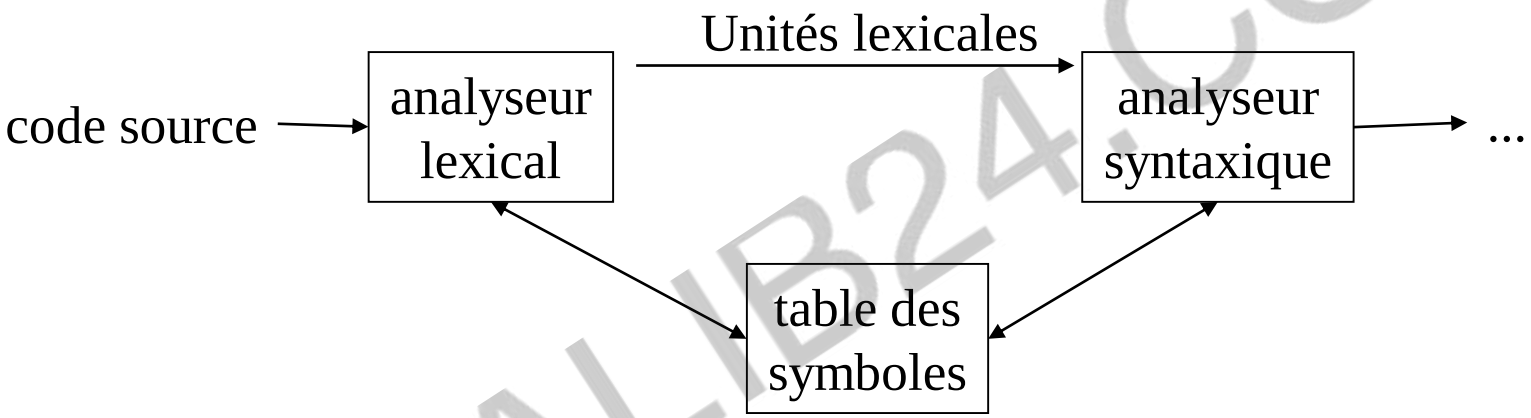
- Les avantages sont multiples :
  - facilité de manipulation de concepts avancés
  - maintenabilité accrue du compilateur
  - portage sur d'autres machines plus aisé

  Par exemple, le compilateur C++ de Björne Stroustrup est écrit en C, ...
- Il est même possible d'écrire un compilateur pour un langage L dans ce langage L (gcc est écrit en C ...!) (bootstrap).

Analyse lexicale

Introduction
Unités lexicales et lexèmes
Langages formels et Expressions régulières
Spécification des unités lexicales
Automates finis
L'outil (f)lex
Mise en œuvre d'un analyseur lexical
Erreurs lexicales

Introduction
Rôle d'un analyseur lexical



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

- En plus, l'analyseur lexical réalise certaines tâches secondaires comme l'élimination de caractères superflus (commentaires, tabulations, fin de lignes, ...), et gère aussi les numéros de ligne dans le programme source pour pouvoir associer à chaque erreur rencontrée par la suite la ligne dans laquelle elle intervient.

Dans ce chapitre, nous abordons des techniques de spécifications
et d'implantation d'analyseurs lexicaux.
Ces techniques peuvent être appliquées à d'autres domaines.

Unités lexicales et lexèmes

- **Définition 2.1**

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

- Exemples :
  - les chaînes ≥ ≤ < > sont des opérateurs relationnels. L'unité lexicale est **OPREL** (par exemple).
  - Les chaînes *toto*, *ind*, *tab*, *ajouter* sont des **identificateurs** (de variables, ou de fonctions).
  - Les chaînes if, else, while sont des **mots clefs**.
  - Les symboles ,.;() sont des **séparateurs**.

Unités lexicales et lexèmes

- **Définition 2.2**
  Un ***modèle*** est une règle associée à une unité lexicale qui décrit l'ensemble des chaînes du programme qui peuvent correspondre à cette unité lexicale.

- **Définition 2.3**
  On appelle ***lexème*** toute suite de caractère du programme source qui concorde avec le modèle d'une unité lexicale.

- Exemples :
  - L'unité lexicale IDENT (identificateurs) en C a pour modèle : toute suite non vide de caractères composée de chiffres, lettres ou du symbole "_" et qui commencent par une lettre. Des exemples de lexèmes pour cette unité lexicale sont : truc, i,a1, ajouter_valeur ...
  - L'unité lexicale NOMBRE (entier signé) a pour modèle : toute suite non vide de chiffres précédée éventuellement d'un seul caractère parmi {+ -} . Lexèmes possibles : -12, 83204, +0 ...
  - L'unité lexicale REEL a pour modèle : tout lexème correspondant à l'unité lexicale NOMBRE suivi éventuellement d'un point et d'une suite (vide ou non) de chiffres, le tout suivi éventuellement du caractère E ou e et d'un lexème correspondant à l'unité lexicale NOMBRE. Cela peut également être un point suivi d'une suite de chiffres, et éventuellement du caractère E ou e et d'un lexème correspondant à l'unité lexicale NOMBRE. Exemples de lexèmes : 12.4, 0.5e3, 10., -4e-1, -.103e+2 ...

Pour décrire le modèle d'une unité lexicale, on utilisera des **expressions régulières**.
Dans ce qui suit nous présentons des définitions de base : **Mots, Langages** et **Expressions régulières**

Qu'est –ce qu'un langage ?

- Nous nous situons à un niveau très élémentaire d'étude des « langages »
- Pour nous, un langage est simplement un ensemble de suites de lettres…
- on dit aussi: mots,
- on dit aussi: chaînes de caractères

- {0, 01, 10, 00, 11, 000, 001, 010, 100, 011, 101, 110, 111} est un langage
- {*la, lala, lalala, lalalala, lalalalala, …, lalalala…la,* ….} est un langage
- {T, ⊥T, ⊥ ⊥T, ⊥ ⊥ ⊥T, …} est un langage
- L'ensemble des phrases grammaticalement correctes du français standard est un langage
- L'ensemble des programmes syntaxiquement corrects écrits en Pascal est un langage

Alphabet, Mots

- alphabet : tout ensemble fini non vide de symboles dont les éléments sont appelés lettres

Remarque:

- dans certaines applications, on remplacera la notion d'alphabet par celle de vocabulaire, un vocabulaire sera un ensemble de mots (cf. plus loin), pris eux-mêmes comme symboles de départ

- {0, 1}
- {a, b}
- {a, b, c}
- {A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z}
- {α, β, χ, δ, ε, ϕ, γ, η, ι, φ, κ, λ, μ, ν, ο, π, θ, ρ, σ, τ, υ, ϖ, ω, ξ, ψ, ζ}

sont des alphabets couramment utilisés

Mots

- Un mot est une suite finie de lettres

- Exemples:
  - 0010111100 est un mot sur {0, 1}
  - aaabbb est un mot sur {a, b}
  - acababbc est un mot sur {a, b, c}
  - BONJOUUR est un mot sur {A, …, Z}
  - λογοσ est un mot sur {α, …, ζ}
  - code ASCII : caractères sont des mots sur { 0, 1 }
  - analyse lexicale : unités lexicales sur caractères
  - analyse syntaxique : programmes sur unités lexicales

Longueur

- BONJOUUR est un mot de longueur 8

- La *longueur* d'un mot *w*, notée |*w*|, est simplement le nombre de symboles qui le composent.

Mot vide

- *Mot Vide*: noté ε ; n'est composé d'aucun caractère;

- Soit A* l'ensemble des mots sur A,

On a toujours: $\varepsilon \in A^*$

- $|\varepsilon| = 0$.

Concaténation

- Soit deux mots s et s’ sur A
- S’ils sont définis par :

$$s = s_1 \ldots s_n \text{ et } s' = s'_1 \ldots s'_m,$$

- On peut définir un nouveau mot, qu’on notera s.s’ (ou ss’) par:
  - $s.s' = s_1 \ldots s_n s'_1 \ldots s'_m$
  - Il s’agit de la concaténation de s et de s’

Exemple

- Sur {a, b, c}, abbac est un mot, qui est désigné par s
- bcca est un autre mot, qui est désigné par s’
- On vérifiera que leur concaténation s.s’ correspond au mot abbacbcca

Propriétés

- La concaténation est associative…*mais non commutative...*
- Elle a un élément neutre : ε
  - $\varepsilon.x = x.\varepsilon = x$
- Conséquence : la concaténation d'une suite vide de mots est le mot vide
- *Lemme de décomposition:*

Pour tout mot u:

- *Soit u = ε*
- *Soit il existe une lettre a et un mot v de longueur |u|-1 tels que u=av ex: u=XML avec a=X et v=ML*

- Puissance: w, $n \geq 0$
  - $w^0 = \varepsilon$
  - $w^{n+1} = w^n w$

Stratification

- Parmi tous les mots sur A, il y a:
  - Les mots de longueur nulle, formant l'ensemble {ε}
  - Les mots de longueur 1, donnant simplement l'ensemble A (on ne fait pas ici de différence entre lettre et mot d'une lettre)
  - Les mots de longueur 2, formant A^2,
  - …
  - Les mots de longueur n, formant A^n
  - etc.

- D’où la formule:
  - $A^* = \{\varepsilon\} \cup A \cup A^2 \cup A^3 \ldots \cup A^n \ldots$
- Ou:

$$A^* = \bigcup_{n=0}^{\infty} A^n$$

  - avec $A^0 = \{\varepsilon\}$

Langages Formels

- Un langage formel sur un alphabet A sera défini simplement comme une partie de A*
- Donc parmi les langages possibles sur A:
  - $\varnothing$ (langage vide)
  - $\{\varepsilon\}$ (langage réduit au mot vide)
  - A* (langage plein)

Opérations sur les langages

- Σ^+, Σ^*

Σ^+ = {w défini sur Σ : $|w| \geq 1$ et $|w|$ fini}

Défini inductivement par :

- tout élément de Σ appartient à Σ^+,
- si $a \in \Sigma$ et $w \in \Sigma^+$ alors $aw \in \Sigma^+$,
- Σ^+ ne contient que des mots obtenus en appliquant un nombre fini de fois les règles 1 et 2.
- Infini si $\Sigma \neq \phi$.

$\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$.

Exemple: Σ = {a, b}

Σ^+ = {a, b, aa, ab, ba, bb, aab, aba, abb, baa, …}.

Σ^* = {ε, a, b, aa, ab, ba, bb, aab, aba, abb, baa, …}.

L, L_1 et L_2 définis sur Σ,

- *Union, Intersection:*

$(L_1 \cup L_2)$
$(L_1 \cap L_2)$
} Définitions ensemblistes

- *Complémentation:* L défini sur Σ,

$\overline{L} = \Sigma^* \setminus L$.

- *Concaténation* (produit)*:* L_1 et L_2 définis sur Σ,

$L_1L_2 = \{w_1w_2 \mid w_1 \in L_1 \text{ et } w_2 \in L_2\}$
$= \{ w \in \Sigma^*, \exists u \in L1, \exists v \in L2, w = u.v \}$

  - Elément neutre: $L_\varepsilon = \{\varepsilon\}$
    $(LL_\varepsilon = L_\varepsilon L = L)$
  - Elément absorbant: langage $\varnothing$
    $(L\varnothing = \varnothing L = \varnothing)$.
  - $L_\varepsilon \neq \varnothing$

Exemple: **Σ={a, b};**

- L_1= {a, b}; L_2= {bac, b, a},
- L_1L_2= {abac, ab, aa, bbac, bb, ba}

Opérations sur les langages

- Puissance: L^n
  - $L^0 = L_\varepsilon$
  - $L^{n+1} = L^n L$ ($L^n = LL\ldots L$ (n fois))

- Fermeture Iterative L* (ou Fermeture de Kleene)

Ensemble des mots résultant d'une concaténation d'un nombre fini de mots de L

- $L^* = L^0 \cup L^1 \cup L^2 \cup \ldots = \cup_{\{i \geq 0\}} L^i$ **(1)**
- $L^* = \{w \in A^*; \exists n \geq 0, \exists w_1, w_2, \ldots, w_n \in L, w = w_1 w_2 \ldots w_n\}$
- $L^+ = L^1 \cup L^2 \cup \ldots = \cup_{\{i \geq 1\}} L^i$
  $= LL^* = L^*L$

Exemple

- Soit L = {ab, ba} sur {a, b}
- L* = {ε, ab, ba, abab, abba, baab, baba, ababab, ababba, abbaab, abbaba, baabab, baabba, babaab, bababa, etc…}

Propriétés

- associativité
  - $A.(B.C) = (A.B).C$
  - $A \cup (B \cup C) = (A \cup B) \cup C$
- distributivité
  - $A.(B \cup C) = A.B \cup A.C$
- autres
  - $A.\varnothing = \varnothing \qquad A \cup \varnothing = A$
  - $A.\{\varepsilon\} = A$
  - $\varnothing^* = \{\varepsilon\}$
  - $\{\varepsilon\} \cup A.A^* = A^*$

$\emptyset^* = \{\varepsilon\}$

- Car l'égalité (1) donnée à la diapo 55 est valable pour tout L, y compris L = ∅

$\{\varepsilon\} \cup A.A^* = A^*$

- Toujours par l'égalité (1) donnée à la diapo 55, que nous réécrivons ici (A étant un langage quelconque):

  $A^* = \{\varepsilon\} \cup A \cup A^2 \cup A^3 \ldots \cup A^n \ldots$

  d'où:

  $A.A^* = A.\{\varepsilon\} \cup A.A \cup A.A^2 \cup A.A^3 \ldots \cup A.A^n \ldots$

  $= A \cup A^2 \cup A^3 \ldots \cup A^n \ldots$

  Donc $\{\varepsilon\} \cup A.A^* = \{\varepsilon\} \cup A \cup A^2 \cup A^3 \ldots \cup A^n \ldots$

  $= A^*$

Langages réguliers

- Un langage régulier sur un alphabet A est
  - ou bien : $\varnothing$
  - ou bien : $\{\varepsilon\}$
  - ou bien : $\{x\}$ pour n'importe quel $x \in A$
- ou bien encore un langage quelconque obtenu à partir de deux langages réguliers par union ou par produit.

  ou un langage quelconque obtenu à partir d'un langage régulier par fermeture de Kleene.

Exemples

- {a}∪{b} est un langage régulier sur {a,b,c}
- ({a}∪{b})* - idem –
- {a}{a}{a} – idem –
- {a}{a}{a} ({a}∪{b})* - idem –
- ({ε}∪{a}{a})* - idem –
- ({ε}∪{a}{a})*{b}{b}{b} ({a}∪{b})*

- {a}∪{b} = {a, b}
- ({a}∪{b})* = {a, b}*
- {a}{a}{a} = {aaa}
- {a}{a}{a} ({a}∪{b})* = ensemble des mots sur {a, b} qui commencent par aaa
- ({ε}∪{a}{a})*= ensemble des mots constitués d'un nombre pair de a y compris 0
- ({ε}∪{a}{a})*{b}{b}{b} ({a}∪{b})* = ensemble des mots sur {a, b} commençant par un nombre pair de a (y compris ε) suivi de trois b

- L’ensemble des représentations en binaire des entiers pairs est un langage régulier sur {0, 1}
  - car il s’écrit: ({0}∪{1})*0
- L’ensemble des mots se terminant en ‘tion’ est un langage régulier sur {a, b, …, z}
  - car il s’écrit: ({a}∪{b}∪{c}∪…{z})*{t}{i}{o}{n}
- L’ensemble des mots de 3 lettres sur {a, b} est un langage régulier
  - car il s’écrit:
    {a}{a}{a}∪{b}{a}{a}∪{a}{b}{a}∪{a}{a}{b}∪
    {a}{b}{b}∪{b}{a}{b}∪{b}{b}{a}∪{b}{b}{b}

Expressions régulières

- Les expressions régulières servent à désigner les langages réguliers
  - $\varnothing$, ε et toute lettre $x \in A$ sont des expressions régulières sur A,
  - Si α et β sont des expressions régulières sur A, alors:
    - $\alpha+\beta$ et $\alpha\beta$ sont des expressions régulières sur A
    - $(\alpha)^*$ est aussi une expression régulière sur A

- a+b désigne {a}∪{b}
- (a+b)* désigne ({a}∪{b})*
- aaa désigne {a}{a}{a}
- aaa(a+b)* désigne {a}{a}{a} ({a}∪{b})*
- (ε+aa)* désigne ({ε}∪{a}{a})*
- (ε+aa)*bbb(a+b)* désigne ({ε}∪{a}{a})*{b}{b}{b} ({a}∪{b})*

- a+b est une expression régulière sur {a,b,c}
- (a+b)* - idem –
- aaa – idem –
- aaa(a+b)* - idem –
- (ε+aa)* - idem –
- (ε+aa)*bbb(a+b)*

Relation de désignation

- Elle est facile à construire:
  - ∅ désigne ∅,
  - ε désigne {ε}
  - et toute lettre x∈A désigne {x}
  - De plus:
    - Si α désigne A et β désigne B, alors
    - α+β (ou α|β) désigne A∪B et αβ désigne A.B
    - et (α)* désigne (A)*

- On retrouve sur les e.r. les propriétés induites par les langages réguliers :
- associativité
  - $\alpha(\beta\gamma) = (\alpha\beta)\gamma$
  - $\alpha+(\beta+\gamma) = (\alpha+\beta)+\gamma$
- distributivité
  - $\alpha(\beta+\gamma) = \alpha\beta+ \alpha\gamma$
- autres
  - $\alpha\varnothing = \varnothing$ $\quad\quad$ $\alpha+\varnothing = \alpha$
  - $\alpha\varepsilon = \alpha$
  - $\varnothing^* = \varepsilon$
  - $\varepsilon+ \alpha\alpha^* = \alpha^*$

Remarques : pour économiser des parenthèses, on conviendra des priorités décroissantes suivantes : *, concaténation, + .
C'est à dire par exemple que $ab^*+c=((a)((b)^*))+(c)$
La concaténation est distributive par rapport à + : c'est à dire que $r(s+t)=rs+rt$ et $(s+t)r=sr+tr$.

Autres propriétés

- $(\alpha+\beta)^* = (\alpha^*+\beta^*)^* = (\alpha^*\beta^*)^* = (\alpha^*\beta)^*\alpha^*$
  $= \alpha^*(\beta\alpha^*)^*$
- $\alpha(\beta\alpha)^* = (\alpha\beta)^*\alpha$

Comment démontrer ces propriétés ?

- $(\alpha+\beta)^* = (\alpha^*+\beta^*)^*$:

un mot de $(\alpha+\beta)^*$ est une suite de α et de β

par exemple: $\alpha\beta\alpha\alpha\beta\beta\alpha\alpha\alpha\beta$

or une telle suite peut toujours être « ré-analysée » en une suite de α^* et de β^*

cf: $(\alpha)(\beta)(\alpha\alpha)(\beta\beta)(\alpha\alpha\alpha)(\beta)$

c 'est donc aussi un mot de $(\alpha^*+\beta^*)^*$

(et réciproquement)

- $(\alpha+\beta)^* = (\alpha^*\beta^*)^*$:
- de même un mot de $(\alpha+\beta)^*$, par exemple:
  $\alpha\beta\alpha\alpha\beta\beta\alpha\alpha\alpha\beta$
  peut toujours être « ré-analysé » en une suite de $\alpha^*\beta^*$
  cf: $(\alpha\beta)(\alpha\alpha\beta\beta)(\alpha\alpha\alpha\beta)$
  c 'est donc aussi un mot de $(\alpha^*\beta^*)^*$
  (et réciproquement)

- $(\alpha+\beta)^* = (\alpha^*\beta)^*\alpha^*$:
- de même un mot de $(\alpha+\beta)^*$, par exemple:
  $\alpha\beta\alpha\alpha\beta\beta\alpha\alpha\alpha\beta\alpha\alpha$
  peut toujours être « ré-analysé » en une suite de $\alpha^*\beta$ suivie d 'une suite de α
  cf: $(\alpha\beta)(\alpha\alpha\beta)(\beta)(\alpha\alpha\alpha\beta)(\alpha\alpha)$
  c 'est donc aussi un mot de $(\alpha^*\beta)^* \alpha^*$
  (et réciproquement)

- $(\alpha+\beta)^* = \alpha^*(\beta\alpha^*)^*$:
- de même un mot de $(\alpha+\beta)^*$, par exemple:
  $\alpha\beta\alpha\alpha\beta\beta\alpha\alpha\alpha\beta\alpha\alpha$
  peut toujours être « ré-analysé » en une suite de α suivie d'une suite de $\beta\alpha^*$
  cf: $(\alpha)(\beta\alpha\alpha)(\beta)(\beta\alpha\alpha\alpha)(\beta\alpha\alpha)$
  c 'est donc aussi un mot de $(\alpha^*\beta)^*\ \alpha^*$
  (et réciproquement)

- $\alpha(\beta\alpha)^* = (\alpha\beta)^*\alpha$:
- un mot de $\alpha(\beta\alpha)^*$ obéit à un schéma:
- $\alpha(\beta\alpha)(\beta\alpha)(\beta\alpha)(\beta\alpha) \ldots (\beta\alpha)$
- un tel schéma peut être ré-analysé en:
- $(\alpha\beta)(\alpha\beta)(\alpha\beta)(\alpha\beta)(\alpha\beta) \ldots (\alpha\beta)\alpha$

Simplification d'expressions régulières

- (b+aa*b)+(b+aa*b)(a+ba*b)*(a+ba*b)
- ((ε+aa*)b)+(b+aa*b)(a+ba*b)*(a+ba*b)
- (a*b)+(b+aa*b)(a+ba*b)*(a+ba*b)
- ε + AA* = A*
- (a*b)+(a*b)(a+ba*b)*(a+ba*b)
- (a*b)(ε+(a+ba*b)*(a+ba*b))
- (a*b)(a+ba*b)*
- (a*b)(a*(ba*b)*)*
- (A+B)* = (A*B*)*
- (a*b)a*(ba*ba*)*
- (A*B*)* = A*(BA*)*
- a*ba*(ba*ba*)*: c 'est le langage des mots ayant un nombre impair de b

Spécification des unités lexicales

- Nous disposons donc, avec les E.R., d'un mécanisme de base pour décrire des untiés lexicales. Par exemple, une ER décrivant les identificateurs en C pourrait être :

- ident = (a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v |w|x|y|z|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q |R|S|T|U |V|W|X|Y|Z) (a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r| s|t|u|v|w|x|y|z|A|B |C|D|E|F|G|H|I|J|K|L|M |N|O |P|Q| R|S|T|U|V|W|X|Y|Z|0|1|2|3|4|5|6|7|8|9|_)*

- C'est illisible ... !

Spécification des unités lexicales

- Alors on s'autorise des **définitions régulières** et le symbole - sur des types ordonnés (lettres, chiffres, ...).

  Une définition régulière est une suite de définitions de la forme :

$$\begin{cases} d_1 = r_1 \\ \dots \\ d_n = r_n \end{cases}$$

où chaque r_i est une expression régulière sur l'alphabet $\Sigma \cup \{d_1,\dots, d_{i-1}\}$., et chaque d_i est un nom différent.

Identificateurs en C

lettre = **A** | **B** | ... | **Z** | **a** | **b** | ... | **z**

chiffre = **0** | **1** | ... | **9**

id = (*lettre* | **_**) (*lettre* | **_** | *chiffre*)*

Nombres sans signe en C

chiffre = **0** | **1** | ... | **9**

chiffres = *chiffre chiffre**

frac = **.** *chiffres*

exp = (**E** | **e**) (+ | **-** | ε) *chiffres* | ε

num = (*chiffres* (*frac* | **.** | ε) | *frac*) *exp*

Les mêmes avec les abréviations

chiffre = **0** | **1** | ... | **9**

chiffres = *chiffre*$^{+}$

frac = **.** *chiffres*

exp = ((**E** | **e**) (+ | -) ? *chiffres*)?

num = (*chiffres* (*frac* | **.**)? | *frac*) *exp*

Définition des espaces blancs

delim = **\b** | **\t** | **\n**

ws = *delim**

Mise en œuvre d'un analyseur lexical
Spécification des unités lexicales

- Dans la théorie des langages, on définit des **automates** qui sont des machines théoriques permettant la reconnaissance de mots. Ces machines ne marchent que pour certains types de langages . En particulier : les langages réguliers.

- On a vu ce qu'était un mot, un langage, une expression régulière, ...

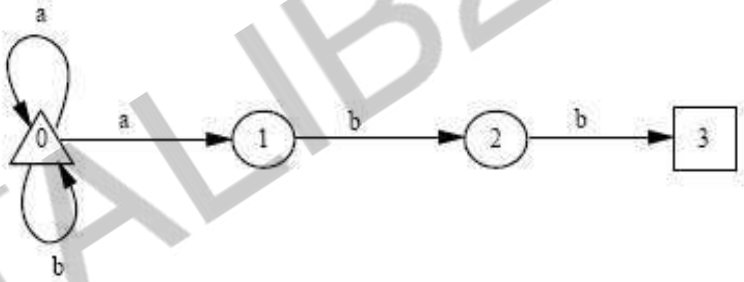
Complétons ces notions de la théorie des langages en étudiant les automates.

Automates à états finis

Définition 2.4 *Un* **automate à états finis (AEF)** *est défini par*

- *un ensemble* **fini** E *d'états*
- *un état* e_0 *distingué comme étant l'***état initial**
- *un ensemble fini* T *d'états distingués comme* **états finaux** *(ou* **états terminaux***)*
- *un alphabet* Σ *des symboles d'entrée*
- *une* **fonction de transition** Δ *qui à tout couple formé d'un état et d'un symbole de* Σ *fait correspondre un ensemble (éventuellement vide) d'états :* $\Delta(e_i, a) = \{e_{i_1}, \ldots, e_{i_n}\}$

Exemple : $\Sigma = \{a, b\}$, $E = \{0, 1, 2, 3\}$, $e_0 = 0$, $T = \{3\}$
$\Delta(0, a) = \{0, 1\}$, $\Delta(0, b) = \{0\}$, $\Delta(1, b) = \{2\}$, $\Delta(2, b) = \{3\}$ (et $\Delta(e, l) = \emptyset$ sinon)
Représentation graphique :



Convention : on dessinera un triangle pour l'état initial et un carré pour les états finaux

Représentation par une **table de transition** :

état	a	b
0	0,1	0
1	-	2
2	-	3
3	-	-

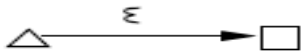
$e_0 = 0$ et $T = \{3\}$

Définition 2.5 *Le* **langage reconnu par un automate** *est l'ensemble des chaînes qui permettent de passer de l'état initial à un état terminal.*

Construction d'un AEF à partir d'une E.R.

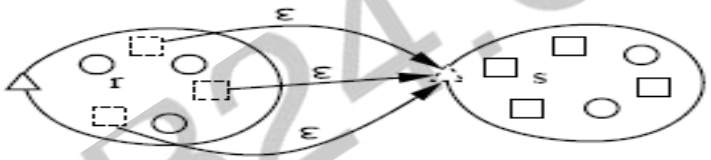
(Construction de Thompson, basée sur la récursivité des expressions régulières)
Pour une expression régulière s, on note $A(s)$ un automate reconnaissant cette expression.

- automate acceptant la chaîne vide



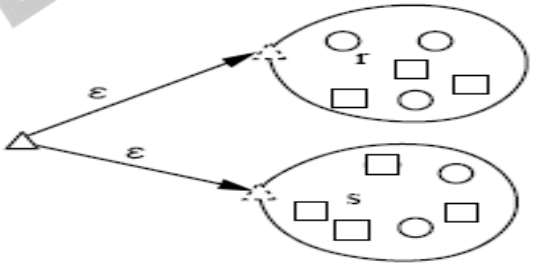
- automate acceptant la lettre a

- automate acceptant $(r)(s)$



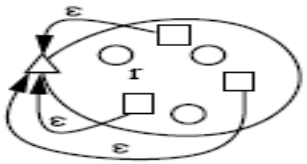
1. mettre une ε-transition de chaque état terminal de $A(r)$ vers l'état initial de $A(s)$
2. les états terminaux de $A(r)$ ne sont plus terminaux
3. le nouvel état initial est celui de $A(r)$
4. (l'ancien état initial de $A(s)$ n'est plus état initial)

- automate reconnaissant $r|s$



1. créer un nouvel état initial q
2. mettre une ε-transition de q vers les états initiaux de $A(r)$ et $A(s)$
3. (les états initiaux de $A(r)$ et $A(s)$ ne sont plus états initiaux)

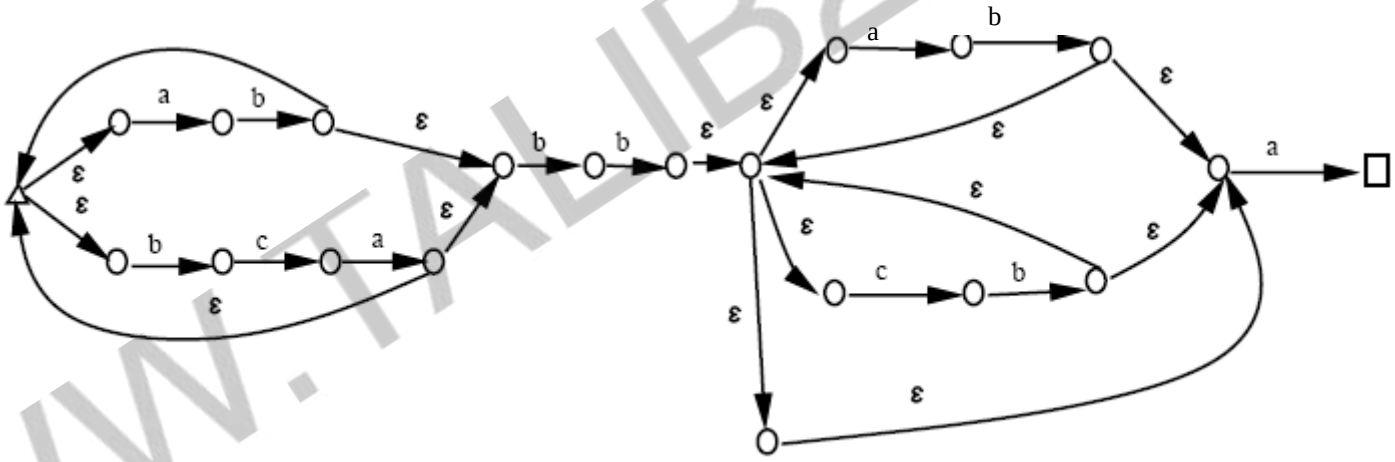
- automate reconnaissant r^+



mettre des ε-transition de chaque état terminal de $A(r)$ vers son état initial

Exemple

- $(ab|bca)+bb(ab|cb)^*a$ correspond à l'automate :

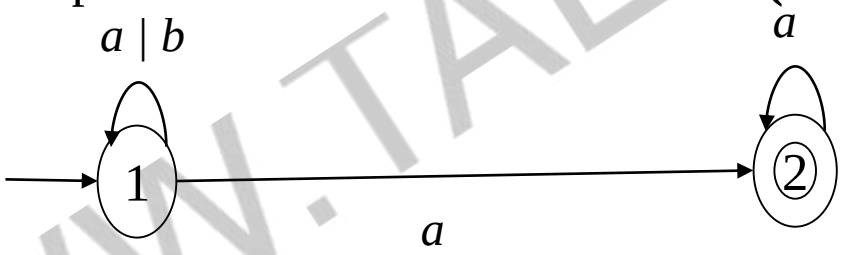


Automates finis déterministes (AFD)

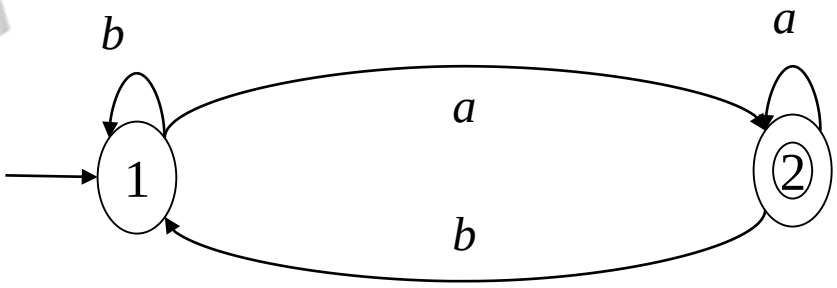
Définition 2.6 On appelle **ε-transition**, une transition par le symbole ε entre deux états.

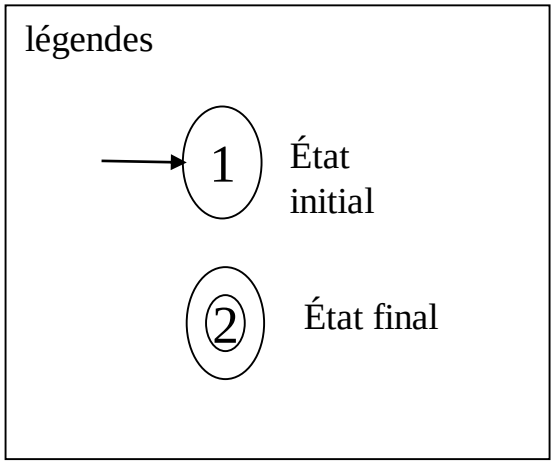
Définition 2.7 Un automate fini est dit **déterministe** lorsqu'il ne possède pas de ε-transition et lorsque pour chaque état *e* et pour chaque symbole *a*, il y a **au plus un** arc étiqueté *a* qui quitte *e*.

Exemple d'automate non déterministe **(AFN)** :



Exemple d'automate déterministe **(AFD)** :





Automates finis déterministes (AFD)

- En général, un automate peut très facilement être simulé par un algorithme (et donc on peut écrire un programme simulant un automate fini).
- Les AFD sont encore plus faciles à simuler (pas de choix dans les transitions, donc jamais de retours en arrière à faire).
- Il existe des algorithmes permettant de **déterminiser** un automate non déterministe (c'est à dire de construire un AFD qui reconnait le même langage que l'AFN donné).
- L'AFD obtenu comporte en général plus d'états que l'AFN, donc le programme le simulant occupe plus de mémoire.

Déterminisation d'un AFN qui ne contient pas d' ε-transitions

Principe : considérer des ensembles d'états plutôt que des états.

1. Partir de l'état initial : $E = \{e_0\}$
2. Construire $E^{(1)}$ l'ensemble des états obtenus à partir de E par la transition a : $E^{(1)} = \Delta(E, a)$
3. Recommencer 2 pour toutes les transitions possibles et pour chaque nouvel ensemble d'état $E^{(i)}$
4. Tous les ensemble d'états $E^{(i)}$ contenant au moins un état terminal deviennent terminaux
5. Renuméroter alors les ensemble d'états en tant que simples états .

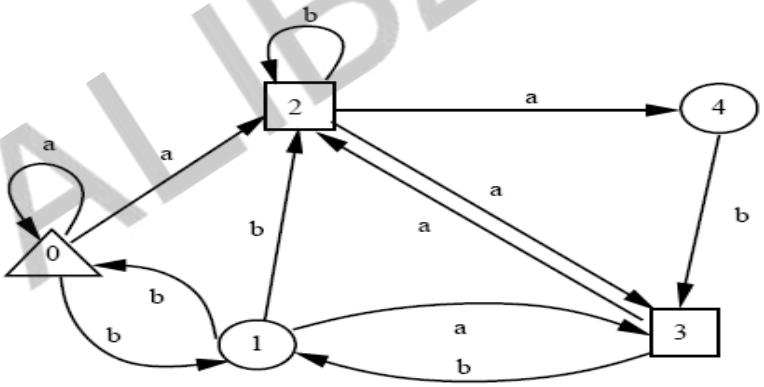


Figure 2.1 Automate non déterministe

Exemple : appliquons l'algorithme sur l'automate représenté figure 2.1 ; dont la table de transition est :

état	a	b
0	0,2	1
1	3	0,2
2	3,4	2
3	2	1
4	-	3

$e_0 = 0$ et $T = \{2, 3\}$

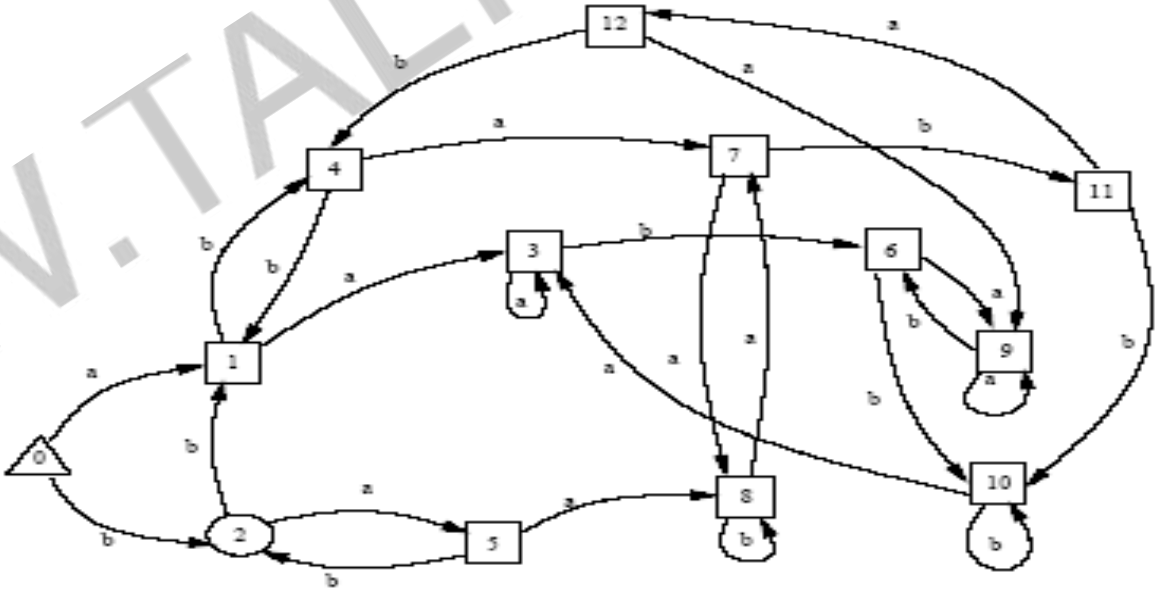
Déterminisation d'un AFN qui ne contient pas d' ε-transitions

0	0,2	1
0,2	0,2,3,4	1,2
1	3	0,2
0,2,3,4	0,2,3,4	1,2,3
1,2	3,4	0,2
3	2	1
1,2,3	2,3,4	0,1,2
3,4	2	1,3
2	3,4	2
2,3,4	2,3,4	1,2,3
0,1,2	0,2,3,4	0,1,2
1,3	2,3	0,1,2
2,3	2,3,4	1,2

=

0	1	2
1	3	4
2	5	1
3	3	6
4	7	1
5	8	2
6	9	10
7	8	11
8	7	8
9	9	6
10	3	10
11	12	10
12	9	4

avec $e_0 = 0$ et $T = \{1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$. On obtient l'automate figure **2.2**



Figur **2.2** Automate déterministe

Déterminisation d'un AFN (cas général)

Principe : considérer l'ε-fermeture des ensembles d'états

Définition 2.8 *On appelle* **ε-fermeture** *de l'ensemble d'états $T = \{e_1, \ldots, e_n\}$ l'ensemble des états accessibles depuis un état e_i de T par des ε-transitions*

ε-fermeture$(\{e_1, \ldots, e_n\}) = \{e_1, \ldots, e_n\} \cup \{e \text{ tq } \exists e_i \text{ avec } i = 1, 2, \ldots, n \text{ tq } e_i \xrightarrow{\varepsilon}\xrightarrow{\varepsilon}\xrightarrow{\varepsilon} \ldots \xrightarrow{\varepsilon} e\}$

Calcul de l'ε-fermeture de $T = \{e_1, \ldots, e_n\}$:

```
Initialiser ε-fermeture(T) à T
Tant que P est non vide faire
    Soit p l'état en sommet de P
    dépiler P
    Pour chaque état e tel qu'il y a une ε-transition entre p et e faire
        Si e n'est pas déja dans ε-fermeture(T)
            ajouter e à ε-fermeture(T)
            empiler e dans P
        finsi
    finpour
fin tantque
```

Exemple. Soit l'AFN

état	a	b	c	ε
0	2	-	0	1
1	3	4	-	-
2	-	-	1,4	0
3	-	1	-	-
4	-	-	3	2

$e_0 = 0$ et $T = \{4\}$

On a ε-fermeture$(\{0\}) = \{0, 1\}$, ε-fermeture$(\{1, 2\}) = \{1, 2, 0\}$, ε-fermeture$(\{3, 4\}) = \{3, 4, 0, 1, 2\}$, ...

Déterminisation d'un AFN (cas général)

Déterminisation :

1. Partir de l'ϵ-fermeture de l'état initial
2. Rajouter dans la table de transition toutes les ϵ-fermetures des nouveaux "états" produits, avec leurs transitions
3. Recommencer 2 jusqu'à ce qu'il n'y ait plus de nouvel "état"
4. Tous les "états" contenant au moins un état terminal deviennent terminaux
5. Renuméroter alors les états.

Sur l'exemple, on obtient

état	a	b	c
0,1	2,3,0,1	4,2,0,1	0,1
0,1,2,3	0,1,2,3	0,1,2,4	0,1,4,2
0,1,2,4	0,1,2,3	0,1,2,4	0,1,3,4,2
0,1,2,3,4	0,1,2,3	0,1,2,4	0,1,2,3,4

=

état	a	b	c
0	1	2	0
1	1	2	2
2	1	2	3
3	1	2	3

avec $e_0 = 0$ et $T = \{2, 3\}$

Minimisation d'un AFD

But : obtenir un automate ayant le minimum d'états possible. En effet, certains états peuvent être équivalents.
Principe : on définit des classes d'équivalence d'états par raffinements successifs. Chaque classe d'équivalence obtenue forme un seul même état du nouvel automate.

1 - Faire deux classes : A contenant les états terminaux et B contenant les états non terminaux.
2 - S'il existe un symbole a et deux états e_1 et e_2 d'une même classe tels que $\Delta(e_1, a)$ et $\Delta(e_2, a)$ n'appartiennent pas à la même classe, alors créer une nouvelle classe et séparer e_1 et e_2.
3 - Recommencer 2 jusqu'à ce qu'il n'y ait plus de classes à séparer.
4 - Chaque classe restante forme un état du nouvel automate

Reprenons l'exemple de la figure 2.2 :

(1) $A = \{0, 2\}$ et $B = \{1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
(2) • $\Delta(0, a)$ et $\Delta(2, a)$ n'appartiennent pas à la même classe, alors A se casse en $A = \{0\}$ et $C = \{2\}$
• $\Delta(5, b)$ et $\Delta(1, b)$ n'appartiennent pas à la même classe, alors B se casse en $B = \{1, 3, 4, 6, 7, 8, 9, 10, 11, 12\}$ et $D = \{5\}$
(3) Ça ne bouge plus.

On obtient donc l'automate à 4 états (correspondants aux 4 classes) de la figure 2.3.
Cet automate reconnait le même langage que l'AFN que nous avions au début.

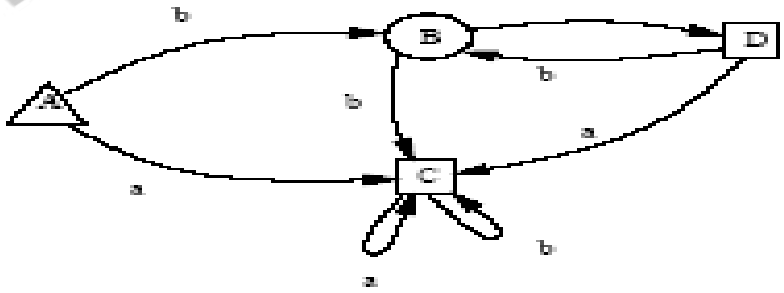


Figure 2.3 : Automate minimum

Calcul d'une E.R. décrite par un A.E.F.

On a dit qu'un AEF et une ER c'était pareil, on a vu comment passer d'une ER à une AEF, maintenant il reste à passer d'un AEF à une ER.

On appelle L_i le langage que reconnaîtrait l'automate si e_i était son état initial. On peut alors écrire un système d'équations liant tous les L_i :

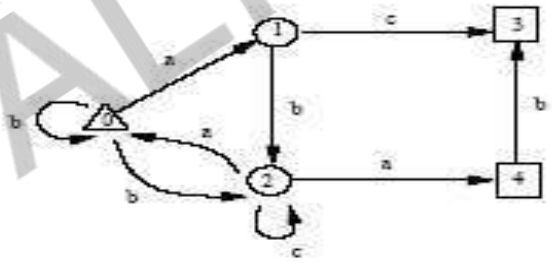
- chaque transition $\Delta(e_i, a) = e_j$ permet d'écrire l'équation $L_i = aL_j$
- pour chaque $e_i \in T$ on a l'équation $L_i = \varepsilon$
- les équations $L_i = \alpha$ et $L_i = \beta$ se regroupent en $L_i = \alpha|\beta$

On résoud ensuite le système (on cherche à calculer L_0) en remarquant juste que

Propriété :
si $L = \alpha L|\beta$ *alors* $L = \alpha^*\beta$

- **Attention :** si on a $L = \alpha L$, ça n'implique **absolument pas** que $L = \alpha^*$. Ca indique juste qu'on est dans une impasse (on boucle) et donc on a dû se planter quelquepart.
- si on a $L = aL|bL|c$, ça s'écrit encore $L = (a|b)L|c$ ce qui implique que $L = (a|b)^*c$. On pourrait faire aussi : $L = aL|bL|c = aL|(bL|c) \Rightarrow L = a^*(bL|c) = (a^*bL)|(a^*c) \Rightarrow L = (a^*b)^*a^*c$ ce qui exprime le même langage mais en moins lisible[2]

Exemple complet : soit l'automate



on obtient le système

$$\begin{cases} L_0 = bL_0|aL_1|bL_2 \\ L_1 = bL_2|cL_3 \\ L_2 = aL_0|cL_2|aL_4 \\ L_3 = \varepsilon \\ L_4 = \varepsilon|bL_3 \end{cases}$$

$$\Leftrightarrow \begin{cases} L_3 = \varepsilon \\ L_4 = \varepsilon|b \\ L_2 = aL_0|cL_2|a(\varepsilon|b) = aL_0|cL_2|a|ab = c^*(a|ab|aL_0) \\ L_1 = bL_2|c = c|bc^*(a|ab|aL_0) \\ L_0 = bL_0|ac|abc^*(a|ab|aL_0)|bc^*(a|ab|aL_0) \end{cases}$$

$$\Rightarrow L_0 = ac|(a|\varepsilon)bc^*(a|ab)|(a|\varepsilon)bc^*aL_0|bL_0$$

$$\Leftrightarrow L_0 = (((a|\varepsilon)bc^*a)|b)^*(ac|(a|\varepsilon)bc^*(a|ab))$$

Programmation d'un AEF

Il est très facile d'écrire un programme simulant un AEF (c'est encore plus facile si c'est un AFD).
Par exemple

```
void Etat1()
{
  char c;
  c= getchar();
  swicth(c) {
    case ':' : Etat2();
               break;
    case '<' : Etat5();
               break;
    case 'a' : Etat57();
               break;
    ...
    default : ERREUR();
  }
 }

 void Etat36524()
 ...
```

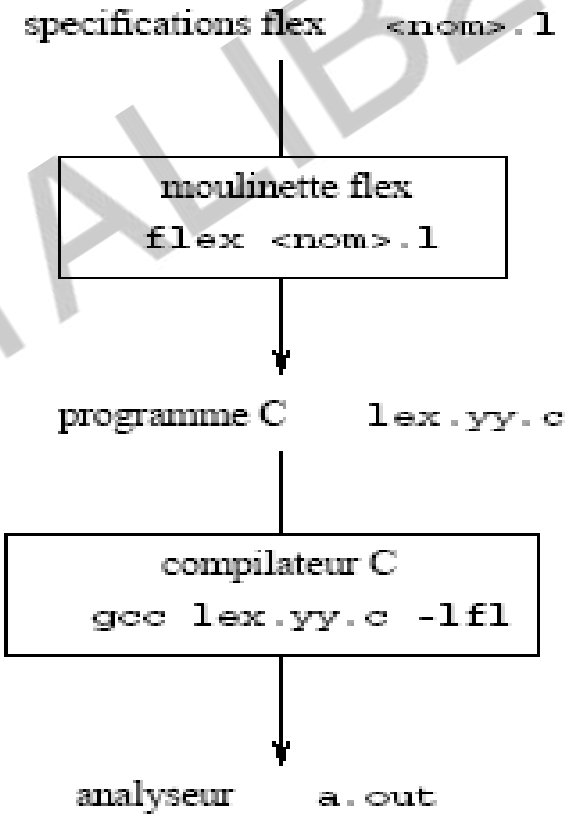
Mais bon ce n'est pas évident du tout dans tout ça de s'y retrouver si on veut modifier l'automate, le compléter, ...
Heureusement, il existe des outils pour écrire des programmes simulant des automates à partir de la simple donnée des expressions régulières. Par exemple : `flex`
Exemple de programme en `flex` :

```
chiffre [0-9]
lettre  [a-zA-Z]
entier  [+-]?[1-9]{chiffre}*
ident   {lettre}({lettre}|{chiffre})*
%%
":="    { return AFF; }
"<="    {
          attribut = INFEG;
          return OPREL;
        }
if|IF|If|iF  {return MC_IF;}
{entier}  {return NB;}
{ident}   {return ID;}
%%
main() {
 yylex();
}
```

L'outil (f)lex

- De nombreux outils ont été batis pour construire des analyseurs lexicaux à partir de notations spécifiques basés sur des expressions régulières.

- (f)lex est un utilitaire d'unix. Son grand frère flex est un produit gnu. (f)lex accepte en entrée des spécifications d'unités lexicales sous forme de définitions régulières et produit un **programme** écrit dans un langage de haut niveau (ici le langage C) qui, une fois compilé, reconnait ces unités lexicales (ce programme est donc un analyseur lexical).



Structure du fichier de spécifications (f)lex

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

%{
Partie I.1 : déclaration (en C) de variables, constantes,. . .
%}
Partie I.2 : déclaration de définitions régulières
%%
Partie II : expressions régulières et actions correspondantes
%%
Partie III : bloc principal (fonction main) et fonctions C supplémentaires

- La partie principale est la partie II. C'est la partie des règles de traduction. Ces règles sont de la forme :

```
exp1 {action1}
exp2 {action2}
```

- Les exp*i* sont des expressions régulières (f)lex et doivent commencer au début de la ligne (en colonne 0).
- Les action*i* sont des blocs d'instructions en C qui doivent commencer sur la même ligne que l'expression correspondante.
- La partie I.1 est facultative (ainsi donc que les %{ et %}et la partie III aussi (ainsi donc que la ligne %% qui la précède).

Les expressions régulières (f)lex

- Une expression régulière (f)lex se compose de caractères normaux et de méta-caractères qui ont une signification spéciale : $, \, ^ , [,], {, }, <, >, +, -, *, /, |, ?.
- Attention, (f)lex fait la différence entre les minuscules et les majuscules.

expression	reconnaît, accepte	exemples
c	tout caractère c qui n'est pas un méta-caractère	`a b 0 ; X`
$r_1 r_2$	r_1 suivie de r_2	`ab bidule truc,machin`
r^*	0 ou plusieurs occurrences de r	`a* ab*c`
r^+	1 ou plusieurs occurrences de r	`a+ ab+c`
(r)	l'expression régulière r	`(ab)+c`
$r_1 \mid r_2$	r_1 ou r_2	`(a|b)* (aa|bc+a)*`
$"s"$	la chaîne de caractère s littéralement	`"abc*+" " x "`
$r?$	0 ou 1 occurrence de r	`("+"|"-")?(0|1|2)+`
$r\{m\}$	m occurrences de r	`a{3}`
$r\{m,n\}$	m à n occurrences de r	`(ab){5,8}`
{}	pour faire référence à une définition régulière	`{lettre} {chiffre}+`
$[s]$	n'importe lequel des caractères constituant la chaîne s	`[abc] [+*,.\ n]`
$c_1 - c_2$	dans des [ ] : un caractère parmis ceux allant de c_1 à c_2	`[a-z] var[0-9]+ [g-m3-6]`
$[\wedge s]$	n'importe lequel des caractères **à l'exception** de ceux constituant la chaîne s	`[^ \t\n] [^aeiouAEIOU]`
$\backslash c$	pour que les méta-caractères redeviennent des simples caractères	`\+`
.	n'importe quel caractère, excepté le caractère $\backslash n$	`a.b [a-z]+.*[23]`
^	**comme premier caractère** de l'expression, signifie le début de la ligne	`^abc`
$	**comme dernier caractère** de l'expression, signifie la fin de la ligne	`abc$`
<<EOF>>	fin de fichier	<<EOF>>

Les expressions régulières (f)lex

- Les méta-caractères $, ^ et / ne peuvent pas apparaître dans des () ni dans des définitions régulières.
- Le méta-caractère ^ perd sa signification « début de ligne »s'il n'est pas au début de l'expression régulière.
- Le méta-caractère $ perd sa signification « fin de ligne »s'il n'est pas à la fin de l'expression régulière.
- A l'intérieur de [], seul \ reste un méta-caractère. Le - ne le reste que s'il n'est ni au début ni à la fin.

- **Priorités :**
  - truc|machin* est interprété comme (truc)|(machi(n*))
  - abc{1,3} est interprété avec lex comme (abc){1,3} et avec flex comme ab(c{1,3})
  - truc|machin est interprété avec lex comme (truc)|machin et avec flex comme (truc|machin)
  - Avec lex, une référence à une définition régulière n'est pas considérée entre {}, en flex si. Par exemple :

```
id [a-z][a-z0-9A-Z]
%%
truc{id}*
```

ne reconnait pas "truc" avec lex, mais le reconnait avec flex.

Variables et fonctions prédéfinies

- **char yytext[ ]** : tableau de caractères qui contient la chaîne d'entrée qui a été acceptée.
- **int yyleng** : longueur de cette chaîne.
- **int yylex()** : fonction qui lance l'analyseur.
- **int main()** : la fonction main() par défaut contient juste un appel à yylex(). L'utilisateur peut la redéfinir dans la section des fonctions auxiliaires.

Options de compilation flex

-d pour un mode debug
-i pour ne pas différencier les majuscules des minuscules
-s pour supprimer l'action par défaut (sortira alors en erreur lors de la rencontre d'un caractère qui ne correspond à aucun motif)

Exemples de fichier .l

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

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

Exemples de fichier .l

- Ce deuxième exemple supprime les lignes qui commencent par p ainsi que tous les entiers, remplace les o par des * et retourne le reste inchangé .

```
chiffre    [0-9]
entier     {chiffre}+
%%
{entier}   printf("je censure les entiers");

^p(.)*\n   printf("je deteste les lignes qui commencent par p\n");

o          printf("*") ;

.          printf("%c",yytext[0]);
```

Mise en œuvre d'analyseur lexical

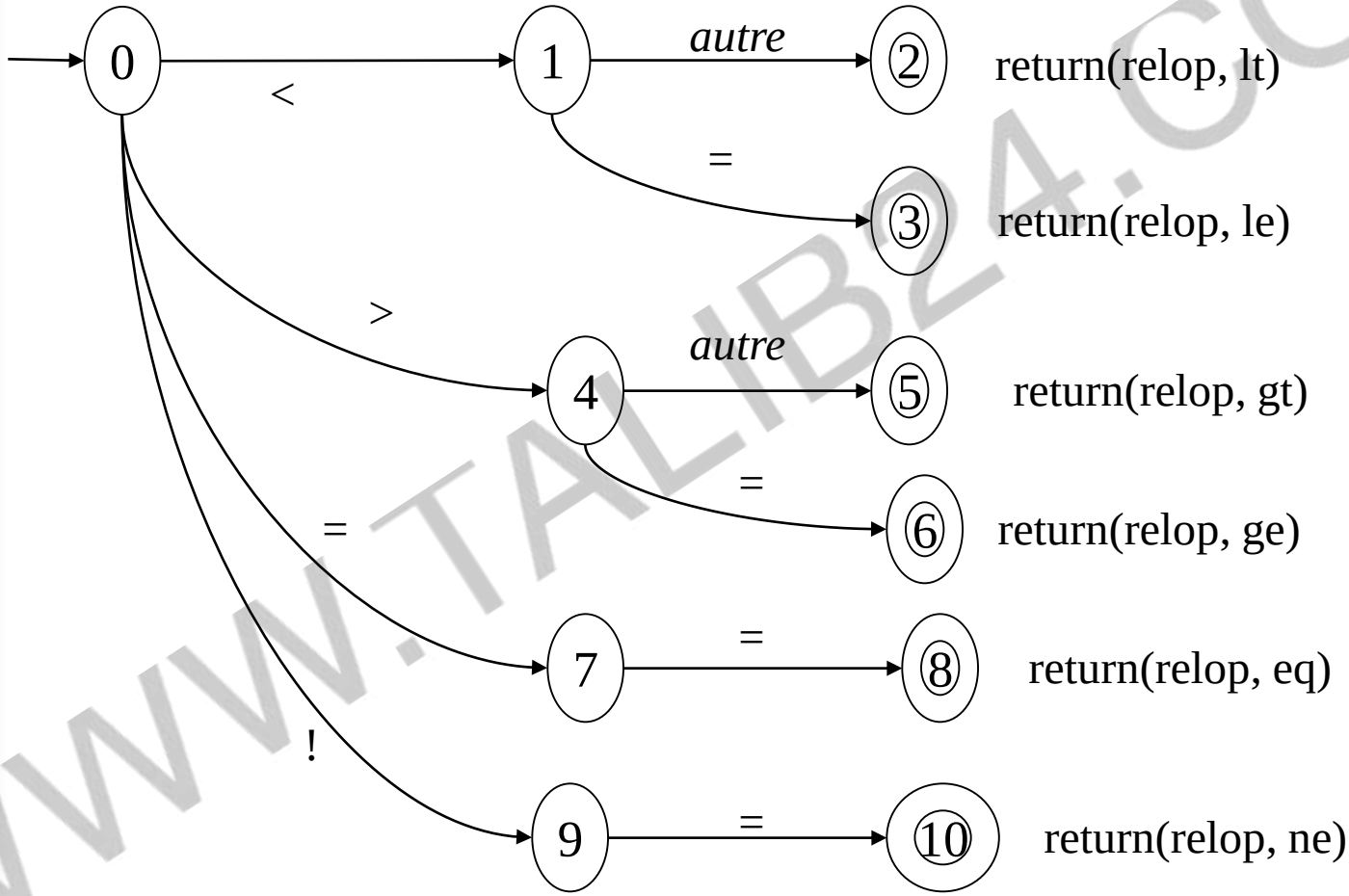
- Récapitulons : le rôle d'un analyseur lexical est la reconnaissance des unités lexicales. Une unité lexicale peut (la plupart du temps) être exprimée sous forme de définitions régulières.
- Dans la théorie des langages, on définit des automates qui sont des machines théoriques permettant la reconnaissance de mots. Ces machines ne marchent que pour certains types de langages . En particulier :

  Théorème 2.1 Un langage régulier est reconnu par un automate fini.

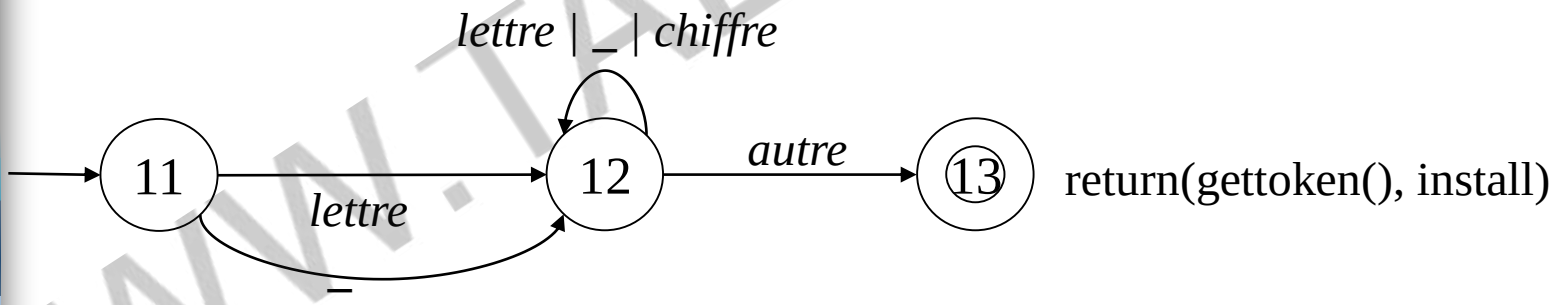
- Pour écrire un analyseur lexical pour un langage régulier, il "suffit" donc d'écrire un programme simulant l'automate reconnaissant ses unités lexicales.

- Lorsqu'une unité lexicale est reconnue, elle envoyée à l'analyseur syntaxique, qui la traite, puis repasse la main à l'analyseur lexical qui lit l'unité lexicale suivante dans le programme source. Et ainsi de suite, jusqu'à tomber sur une erreur ou jusqu'à ce que le programme source soit traité en entier.

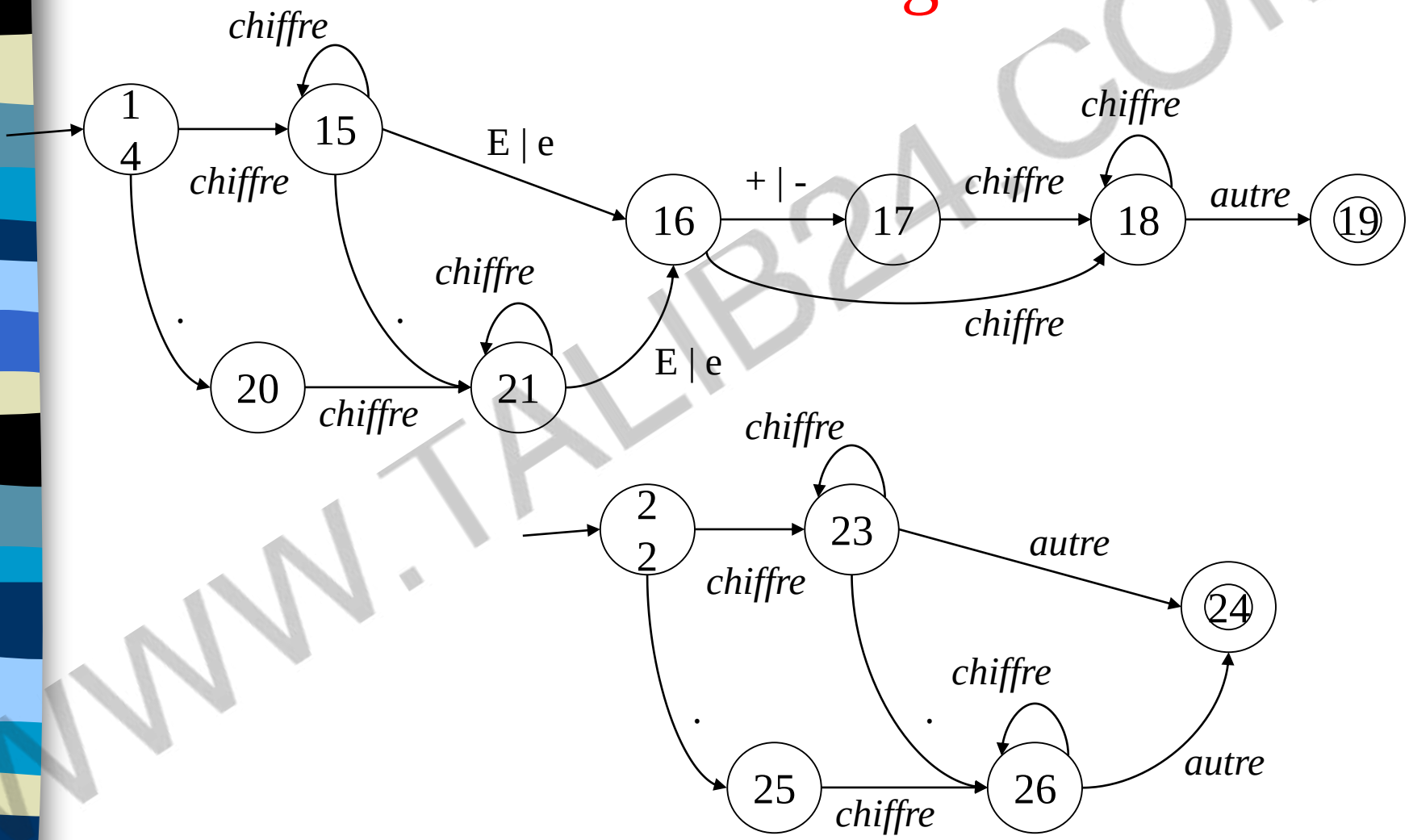
Exemple
Opérateurs de comparaison



Exemple
Identificateurs en C



Exemple
Nombres sans signe en C



Exemple
Implémentation en C

Un programme C qui reconnaît les unités lexicales définies par les automates au dessus.

```
/* Recherche de l'état initial du prochain automate */
int state = 0, start = 0 ;
int lexical_value ;
int fail() {
    forward = token_debut ;
    switch(start) {
        case 0 : start = 11 ; break ;
        case 11 : start = 14 ; break ;
        case 14 : start = 22 ; break ;
        case 22 : recover() ; break ;
        default : /* erreur */ }
    return start ; }
```

```
token nexttoken() {
    while (1) {
        switch(state) {
            case 0 : c = nextchar() ;
            if (c==blank||c==tab||c==newline) {
                state = 0 ; debut ++ ; }
            else if (c == '<') state = 1 ;
            else if (c == '>') state = 4 ;
            else if (c == '=') state = 7 ;
            else if (c == '!') state = 9 ;
            else state = fail() ;
            break ;
        /* ... cas 1 - 10 ... */
            case 11 : c = nextchar() ;
            if (isletter(c)||c=='_') state = 12 ;
            else state = fail() ;
            break ;
```

```
                case 12 : c = nextchar() ;
                if (isletter(c)||c=='_') state = 12 ;
                else if (isdigit(c)) state = 12 ;
                else state = 13 ;
                break ;
                case 13 : retract(1) ; install_id() ;
                return(gettoken()) ;
        /* ... cas 14-26 ... */
                }
        }
}
```

Erreurs lexicales

- Peu d'erreurs sont détectables au niveau lexical, car un analyseur lexical a une vision très locale du programme source. Les erreurs se produisent lorsque l'analyseur est confronté à une suite de caractères qui ne correspond à aucun des modèles d'unité lexicale qu'il a à sa disposition.

- Par exemple, dans un programme C, si l'analyseur rencontre esle, s'agit t-il du mot clef else mal orthographié ou d'un identificateur ? Dans ce cas précis, comme la chaîne correspond au modèle de l'unité lexicale IDENT (identificateur), l'analyseur lexical ne détectera pas d'erreur, et transmettra à l'analyseur syntaxique qu'il a reconnu un IDENT. S'il s'agit du mot clef mal orthographié, c'est l'analyseur syntaxique qui détectera alors une erreur.

- Autre exemple : s'il rencontre 1i, il ne sait pas s'il s'agit du chiffre 1 suivi de l'identificateur i ou d'une erreur de frappe (et dans ce cas l'utilisateur pensait-il à i1, i, 1 ou à tout autre chose ?). L'analyseur lexical peut juste signaler que la chaîne 1i ne correspond à aucune unité lexicale définie.

- Lorsque l'analyseur est confronté à une suite de caractères qui ne correspond à aucun de ses modèles, plusieurs stratégies sont possibles :

  - **mode panique** : on ignore les caractères qui posent problème et on continue.

  - transformations du texte source : insérer un caractère, remplacer, échanger, ...
    La correction d'erreur par transformations du texte source se fait en calculant le nombre minimum de transformations à apporter au mot qui pose problème pour en obtenir un qui ne pose plus de problèmes. On utilise des techniques de calcul de distance minimale entre des mots. Cette technique de récupération d'erreur est très peu utilisée en pratique car elle est trop couteuse à implanter. En outre, au niveau lexical, elle est peu efficace, car elle entraine d'autres erreurs lors des phases d'analyse suivantes.

  - La stratégie la plus simple est le mode panique. En fait, en général, cette technique se contente de refiler le problème à l'analyseur syntaxique. Mais c'est efficace puisque c'est lui, la plupart du temps, le plus apte à le résoudre.

COMPILATION
ET
THEORIE DES LANGAGES

Analyse syntaxique

Introduction

Syntaxe et grammaires

Analyse descendante

Analyse ascendante

Traitement des erreurs

Bison

Introduction

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

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

Le problème est donc :

- étant donnée une grammaire G
- étant donné un mot m (un programme)

→ est ce que *m* appartient au langage généré par *G* ?

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

Syntaxe

Syntaxe

Contraintes sur l'écriture du code dans les langages de programmation

Sémantique

Interprétation du code

Règles de grammaire

Servent à spécifier la syntaxe

symbole --> expression

Dans l'expression on peut avoir deux sortes de symboles :

- ceux du langage final : les symboles **terminaux** (les mêmes qu'en analyse lexicale)
- des symboles intermédiaires, les **variables** ou **non-terminaux**

Exemple

Grammaire pour les expressions arithmétiques simples

E --> **nombre**

E --> **(** *E* **)**

E --> *E* + *E*

E --> *E* **-** *E*

E --> *E* ***** *E*

E --> *E* / *E*

Le non-terminal *E* désigne les expressions

Le symbole terminal **nombre** représente les chaînes de caractères qui sont des nombres

Les autres symboles terminaux sont **( ) + - * /**

Définition formelle

Une grammaire est définie par un quadruplet $<A,V,P,S>$:

- un alphabet ***A*** de symboles terminaux
- un ensemble ***V*** de non-terminaux
- un ensemble fini de règles ***P***

  $(x, w) \in V \times (A|V)^*$ notées $x \dashrightarrow w$
- un non-terminal ***S*** appelé axiome

Un mot sur A est une suite d'éléments de A

A^* est l'ensemble des mots sur A

Un langage formel est une partie de A^*

Dérivations

Si x --> w est une règle de la grammaire, en **remplaçant *x* par *w*** dans un mot on obtient une dérivation :

E*(E) --> E*(E+E)

Enchaînement de dérivations

Si u_0 --> u_1 ... --> u_n on écrit :

$$u_0 \xrightarrow{*} u_n$$

Langage engendré par une grammaire

On s'intéresse aux dérivations qui vont de S à des mots de A^*

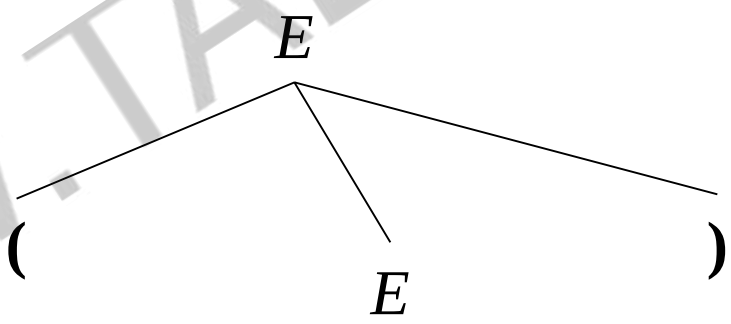
$$L = \{u \in A^* \mid S \xrightarrow{*} u\}$$

Exemple : $E \xrightarrow{*}$ **nombre * (nombre + nombre)**

Arbres

On représente les règles sous forme d'arbres

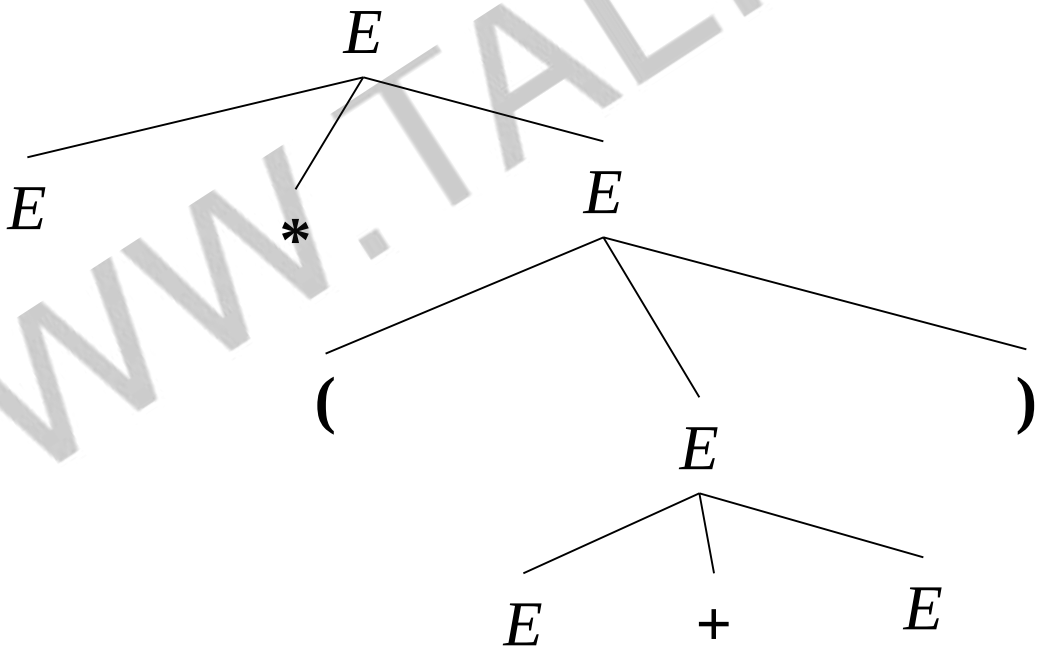
E --> (*E*)



Arbres

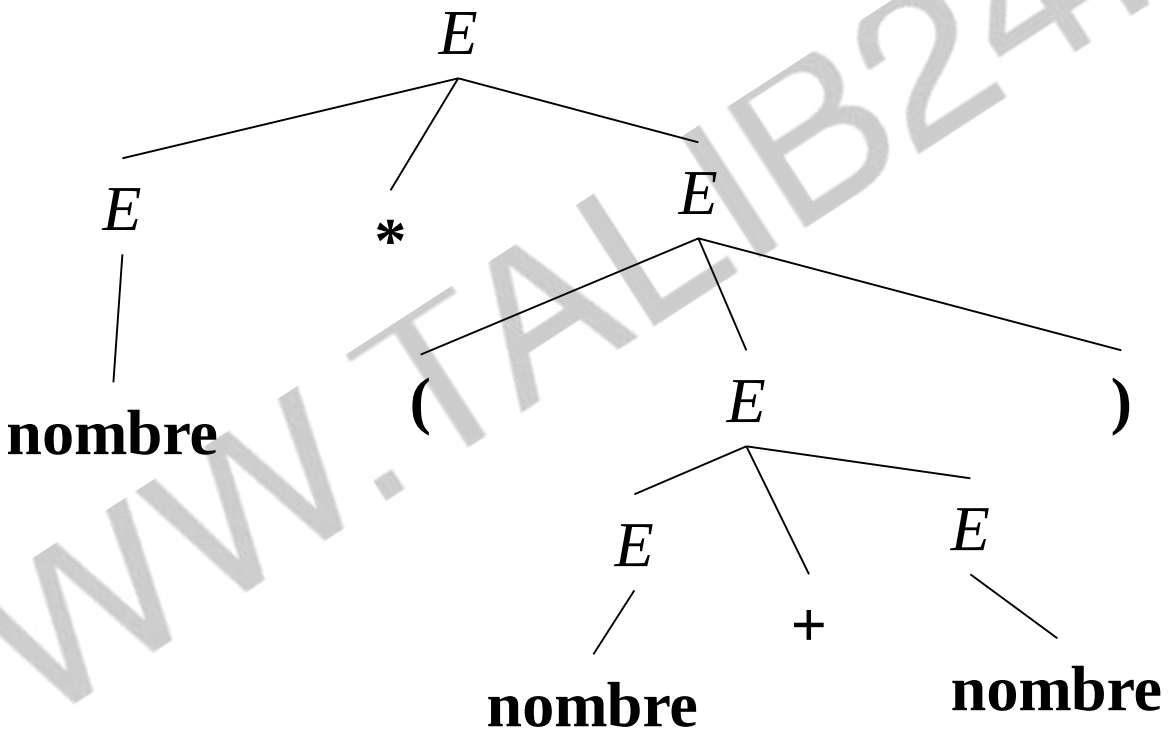
En enchaînant plusieurs dérivations, on a un arbre de hauteur supérieure à 1

$$E \dashrightarrow E * E \dashrightarrow E * (E) \dashrightarrow E * (E + E)$$



Arbres de dérivation

On s'intéresse aux arbres dont la racine est l'axiome et dont toutes les feuilles sont des symboles terminaux



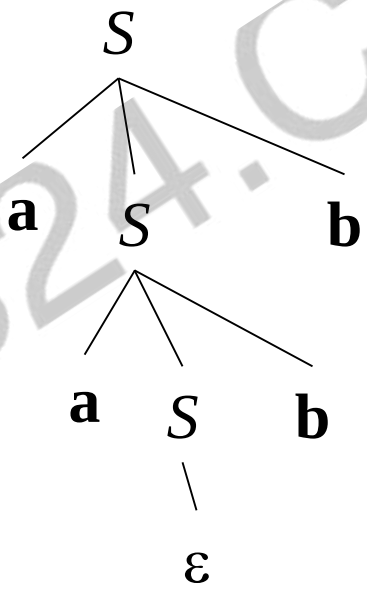
Le langage engendré par la grammaire est l'ensemble des frontières des arbres de dérivation

Arbres de dérivation

S --> **a** S **b**

S --> ε

Quels sont tous les arbres de dérivation pour cette grammaire ?



Les arbres de dérivation de hauteur $n > 0$ obtenus en utilisant $n - 1$ fois la première règle et 1 fois la deuxième

La frontière d'un tel arbre est $\mathbf{a}^{n-1}\mathbf{b}^{n-1}$

Langues naturelles

On peut utiliser les grammaires pour décrire la syntaxe des langues naturelles

<phrase> --> <sujet> <verbe>

<phrase> --> <sujet> <verbe> <complement>

<phrase> --> <sujet> <verbe> <complement> <complement>

<sujet> --> <det> <nom>

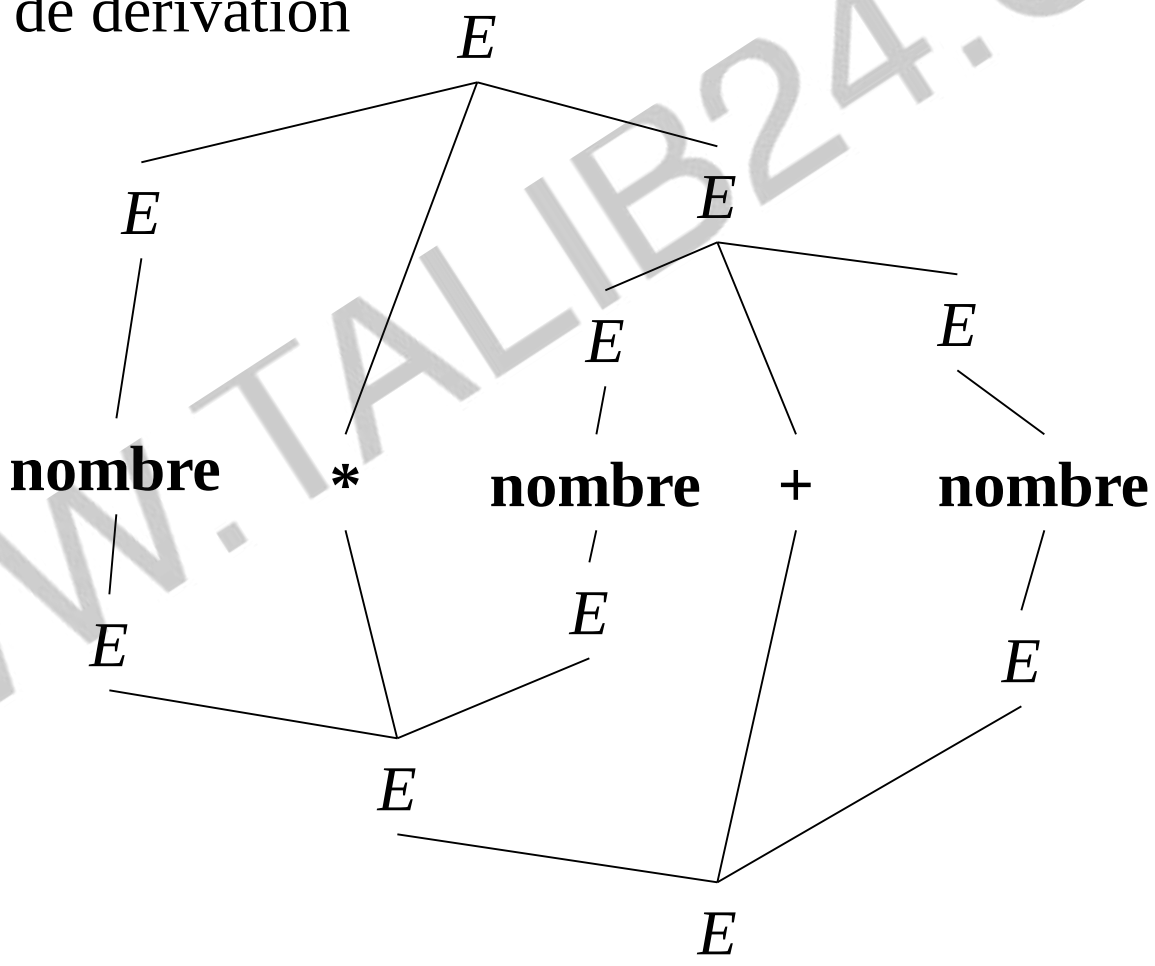
<sujet> --> <det> <adj> <nom>

...

Ce sont les grammaires de Chomsky.

Ambiguïté

Grammaire ambiguë : un même mot possède plusieurs arbres de dérivation



Ambiguïté

Exemple classique de grammaire ambiguë

inst --> **si** *cond* **alors** *inst*

inst --> **si** *cond* **alors** *inst* **sinon** *inst*

inst --> ...

cond --> ...

Exercice : quels sont les arbres pour

Conventions de notation

On regroupe plusieurs règles qui ont le même membre gauche

E --> *E* + *E*

E --> **nb**

E --> *E* + *E* | **nb**

On donne la liste des règles en commençant par l'axiome

inst --> **si** *cond* **alors** *inst*

| **si** *cond* **alors** *inst* **sinon** *inst*

| ...

cond --> ...

Exemples

E --> *E* + *E* | **nb**

Grammaire ambiguë

P --> **(** *P* **)** | ε

Compte des empilements et dépilements (non ambiguë)

P --> **(** *P* **)** *P* | ε

Grammaire des expressions parenthésées (non ambiguë)

E --> *E E* **+** | **nb**

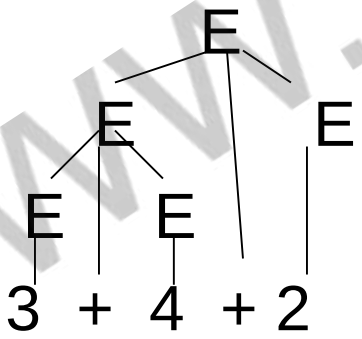
Expressions additives en notation postfixe (non ambiguë)

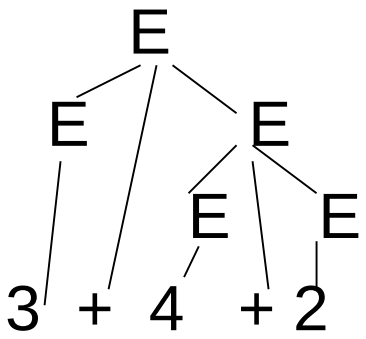
Ambiguité : grammaire pour les expressions

La grammaire suivante des expressions de Pascal avec +,* :

E --> E+E | E*E | (E) | **nb**

est ambiguë : par exemple, 3+4+2 a deux arbres d'analyse :





On notera ces arbres (3+4)+2 et 3+(4+2).

Résolution des ambigüités

Une ambigüité (3+4)+2 vs. 3+(4+2) est appellée une **ambigüité d'associativité** :

on la résout en choisissant le coté récursif (coté gauche en Pascal)

E --> E+F | F

F --> nb | (E)

Résolution des ambigüités

Une ambigüité (3+4)*2 vs. 3+(4*2) est appellée une **ambigüité de précédence** :

on la résoud en ajoutant un nouveau non-terminal par niveau de précédence.

Ici, terme T et facteur F:

E --> E+T | T

T --> T*F | F

F --> nb | (E)

Une grammaire non ambiguë pour les expressions

Cette grammaire force les choix suivants :

- associativité à gauche (c'est-à-dire de gauche à droite)
- priorité de * et / sur + et -.

Pour cela, elle utilise trois niveaux : *E*, *T*, *F*, au lieu d'un seul

- *F* (facteur) : expression qui n'est pas une opération
- *T* (terme) : fait de facteurs avec éventuellement * ou /
- *E* (expression) : fait de termes avec éventuellement + ou -

E --> *E* **+** *T* | *E* **-** *T* | *T*

T --> *T* ***** *F* | *T* / *F* | *F*

F --> **(** *E* **)** | **nb**

Grammaires équivalentes

Deux grammaires sont équivalentes si elles engendrent le même langage

E --> E **+** T | E **-** T | T

T --> T ***** F | T / F | F

F --> **(** E **)** | **nb**

E --> $T\ E'$

E' --> **+** $T\ E'$ | **-** $T\ E'$ | ε

T --> $F\ T'$

T' --> ***** $F\ T'$ | / $F\ T'$ | ε

F --> **(** E **)** | **nb**

Analyse syntaxique

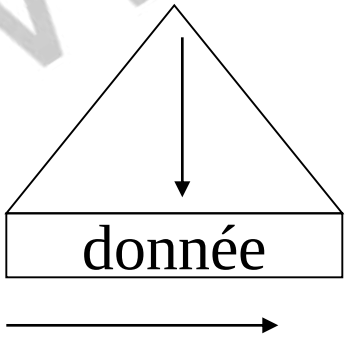
Objectif

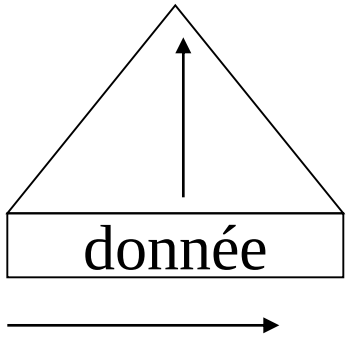
Construire l'arbre de dérivation d'un mot donné

Méthodes

Analyse descendante (descente récursive) : on parcourt le mot en appelant des procédures pour chaque non-terminal

Analyse ascendante : on parcourt le mot en empilant les symboles identifiés





Analyse descendante

Exemple : traduction des expressions arithmétiques par descente récursive.

Cette technique n'est pas applicable à la grammaire suivante :

E --> E **+** T | E **-** T | T

T --> T ***** F | T / F | F

F --> **(** E **)** | **nb**

En raison de la **récursivité à gauche,** la fonction expr() s'appellerait elle-même sans consommer de lexèmes et bouclerait

E --> $T\ E'$

E' --> **+** $T\ E'$ | **-** $T\ E'$ | ε

T --> $F\ T'$

T' --> ***** $F\ T'$ | / $F\ T'$ | ε

F --> **(** E **)** | **nb**

Cette grammaire est récursive à droite : pas de problème

Suppression de la récursivité à gauche

Lemme d'Arden

$X \dashrightarrow Xu \mid v$ $\qquad$ $X \dashrightarrow vX'$

$X' \dashrightarrow uX' \mid \varepsilon$

Dans les deux cas, $X \overset{*}{\dashrightarrow} vu^*$

Prédiction : Premier et Suivant

Pour l'analyse descendante et l'analyse ascendante, on doit prédire quelle règle appliquer en fonction du prochain lexème

Pour cela on définit deux fonctions

Premier() de $(A \mid V)^*$ dans $A \cup \{\varepsilon\}$

$a \in A$: a est dans Premier(u) si on peut dériver à partir de u un mot commençant par a

ε est dans Premier(u) si on peut dériver ε à partir de u

Suivant() de V dans A

$a \in A$: a est dans Suivant(X) si on peut dériver à partir de l'axiome un mot dans lequel X est suivi de a

Prédiction : Premier et Suivant

E --> E **+** T | E **-** T | T

T --> T ***** F | T **/** F | F

F --> **(** E **)** | **nb**

Premier()

(est dans Premier(T) car T --> F --> **(** E **)**

Suivant()

) est dans Suivant(T) car E --> T --> F --> **(** E **)** --> **(** E + T **)**

Calcul de Premier et Suivant

Algorithme

On construit deux graphes dont les sommets sont des symboles et ε

On a un chemin de X à a (ou ε) ssi a (ou ε) est dans Premier(X) (ou Suivant(X))

Premier()

On a un arc de X vers $Y \in A \cup V$ ssi il existe une règle $X \dashrightarrow uYv$ avec $u \overset{*}{\dashrightarrow} \varepsilon$

On a un arc de X vers ε ssi $X \overset{*}{\dashrightarrow} \varepsilon$

Suivant()

On a un arc de X vers a ssi il existe une règle $Y \dashrightarrow uXv$ avec $a \in$ Premier(v)

On a un arc de X vers Y ssi il existe une règle $Y \dashrightarrow uXv$ avec $v \overset{*}{\dashrightarrow} \varepsilon$

(attention, l'arc remonte la flèche de dérivation, car Suivant(Y)⊆Suivant(X))

Exemple

S --> E **$**

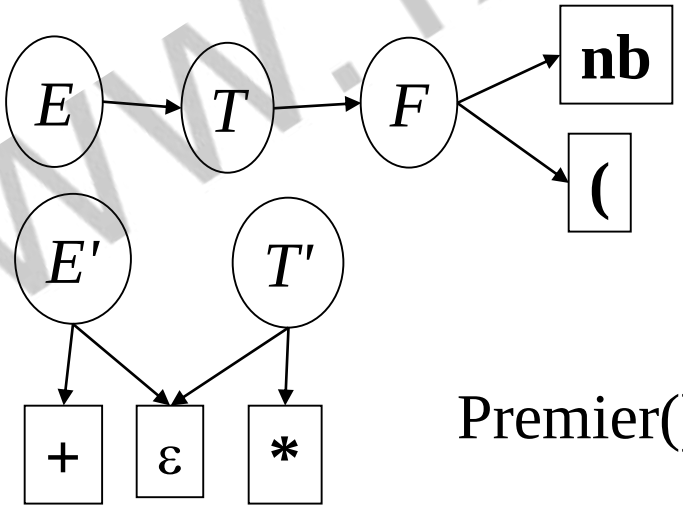
E --> $T\ E'$

E' --> **+** $T\ E'$ | ε

T --> $F\ T'$

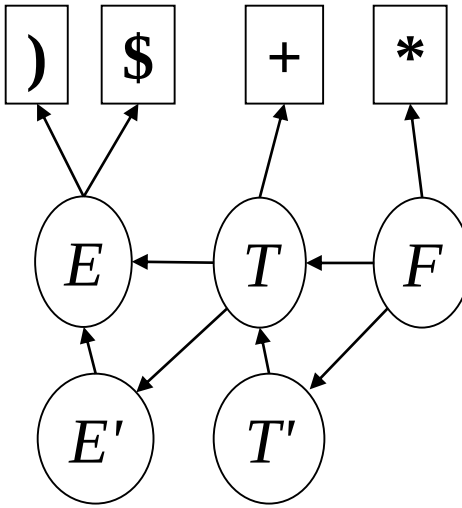
T' --> ***** $F\ T'$ | ε

F --> **(** E **)** | **nb**



Premier()

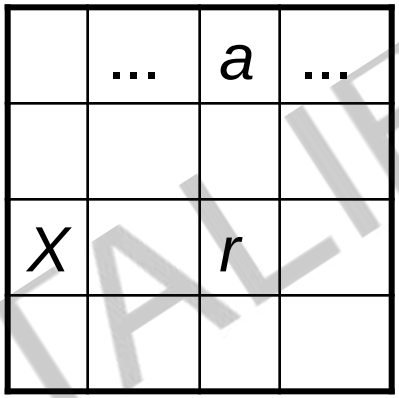
Suivant()



Analyse LL

On utilise Premier() et Suivant() pour construire une table d'analyse.

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



La règle *r* est

- soit X --> u et a est dans Premier(u)
- soit X --> ε et a est dans Suivant (X)

Il y a un conflit s'il y a deux règles dans la même case du tableau

La grammaire est LL(1) s'il n'y a pas de conflits

Exemple

	+	*	(	)	nb	$
E			1		1	
E'	2			3		3
T			4		4	
T'	6	5		6		6
F			7		8	

1 $E \rightarrow T\ E'$

2 $E' \rightarrow +\ T\ E'$

3 $E' \rightarrow \varepsilon$

4 $T \rightarrow F\ T'$

5 $T' \rightarrow *\ F\ T'$

6 $T' \rightarrow \varepsilon$

7 $F \rightarrow (\ E\ )$

8 $F \rightarrow \mathbf{nb}$

Analyse syntaxique

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

Algorithme

données : mot m terminé par \$, table d'analyse M
initialisation de la pile :

S
\$

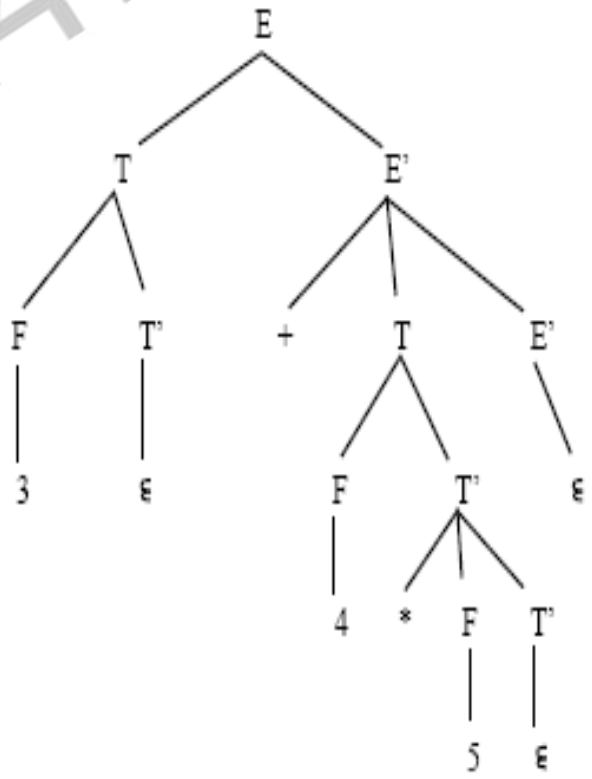
et un pointeur ps sur la 1ère lettre de m

repeter
  Soit X le symbole en sommet de pile
  Soit a la lettre pointée par ps
  Si X est un non terminal **alors**
    Si $M[X, a] = X \rightarrow Y_1 \dots Y_n$ **alors**
      enlever X de la pile
      mettre Y_n puis Y_{n-1} puis ...puis Y_1 dans la pile
      émettre en sortie la production $X \rightarrow Y_1 \dots Y_n$
    sinon (case vide dans la table)
      ERREUR
    finsi
  Sinon
    Si $X =$ \$ **alors**
      Si $a =$ \$ **alors** ACCEPTER
      Sinon ERREUR
      finsi
    Sinon
      Si $X = a$ **alors**
        enlever X de la pile
        avancer ps
      sinon
        ERREUR
      finsi
    finsi
  finsi
jusqu'à ERREUR ou ACCEPTER\$

Exemple

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

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



Exemple

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

PILE	Entrée	Sortie
$\$\ E$	$(7+3)5\$$	$E \to TE'$
$\$\ E'T$	$(7+3)5\$$	$T \to FT'$
$\$\ E'T'F$	$(7+3)5\$$	$F \to (E)$
$\$\ E'T')E($	$(7+3)5\$$	
$\$\ E'T')E$	$7+3)5\$$	$E \to TE'$
$\$\ E'T')E'T$	$7+3)5\$$	$T \to FT'$
$\$\ E'T')E'T'F$	$7+3)5\$$	$F \to 7$
$\$\ E'T')E'T'7$	$7+3)5\$$	
$\$\ E'T')E'T'$	$+3)5\$$	$T' \to \varepsilon$
$\$\ E'T')E'$	$+3)5\$$	$E' \to$ * T E'
$\$\ E'T')E'T+$	$+3)5\$$	
$\$\ E'T')E'T$	$3)5\$$	$T \to FT'$
$\$\ E'T')E'T'F$	$3)5\$$	$F \to 3$
$\$\ E'T')E'T'3$	$3)5\$$	
$\$\ E'T')E'T'$	$)5\$$	$T' \to \varepsilon$
$\$\ E'T')E'$	$)5\$$	$E' \to \varepsilon$
$\$\ E'T')$	$)5\$$	
$\$\ E'T'$	$5\$$	**ERREUR!!**

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

Grammaires LL(1)

L'algorithme précédent ne peut pas être appliqué à toutes les grammaires. En effet, si la table d'analyse comporte des entrées multiples (plusieurs productions pour une même case $M[A, a]$), on ne pourra pas faire une telle analyse descendante car on ne pourra pas savoir quelle production appliquer.

Définition *On appelle* **grammaire LL(1)** *une grammaire pour laquelle la table d'analyse décrite précédemment n'a aucune case définie de façon multiple.*

Le terme "LL(1)" signifie que l'on parcourt l'entrée de gauche à droite (L pour *Left to right scanning*), que l'on utilise les dérivations gauches (L pour *Leftmost derivation*), et qu'**un seul** symbole de pré-vision est nécessaire à chaque étape nécessitant la prise d'une décision d'action d'analyse (1).

Par exemple,

$$\begin{cases} S \to aAb \\ A \to cd|c \end{cases}$$

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

	a	c	b	d	$\$$
S	$S \to aAb$				
A		$A \to cd$ $A \to c$			

Grammaires LL(1)

Il y a deux réductions pour la case $M[A, c]$, donc ce n'est pas une grammaire LL(1). On ne peut pas utiliser cette méthode d'analyse. En effet, on l'a déja remarqué, pour pouvoir choisir entre la production $A \to cd$ et la production $A \to c$, il faut lire la lettre qui suit celle que l'on pointe (donc **deux** symboles de pré-vision sont nécessaires) .

Autre exemple :

$$\begin{cases} S \to aTbbbb \\ T \to Tb|\varepsilon \end{cases}$$ n'est pas LL(1)

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

Récursivité à gauche immédiate

Définition : Une grammaire est **immédiatement récursive à gauche** si elle contient un non-terminal *A* tel qu'il existe une production *A* -->*Aα* où *α* est une chaîne quelconque.

Exemple :

$$\begin{cases} S \to ScA|B \\ A \to Aa|\varepsilon \\ B \to Bb|d|e \end{cases}$$

Cette grammaire contient plusieurs récursivités à gauche immédiates.

Elimination de la récursivité à gauche immédiate :

Remplacer toute règle de la forme $A \to A\alpha|\beta$ par

$$A \to \beta A'$$
$$A' \to \alpha A'|\varepsilon$$

Récursivité à gauche immédiate

Théorème La grammaire ainsi obtenue reconnait le même langage que la grammaire initiale.

Sur l'exemple, on obtient :

$$\begin{cases} S \to BS' \\ S' \to cAS'|\varepsilon \\ A \to A' \\ A' \to aA'|\varepsilon \\ B \to dB'|eB' \\ B' \to bB'|\varepsilon \end{cases}$$

Cette grammaire reconnait le même langage que la première.

Par exemple, le mot *dbbcaa* s'obtient à partir de la première grammaire par les dérivations

$$S \to ScA \to BcA \to BbcA \to BbbcA \to dbbcA \to dbbcAc \to dbbcAaa \to dbbcaa.$$

Il s'obtient à partir de la deuxième grammaire par

$$S \to BS' \to dB'S' \to dbB'S' \to dbbB'S' \to dbbS' \to dbbcAS' \to dbbcA'S' \to dbbcaA'S' \to dbbcaaA'S' \to dbbcaaS' \to dbbcaa.$$

Récursivité à gauche non immédiate

Définition Une grammaire est **récursive à gauche** si elle contient un non-terminal *A* tel qu'il existe une dérivation $A \xrightarrow{+} A\alpha$ où *α* est une chaîne quelconque.

Exemple : $\begin{cases} S \rightarrow Aa|b \\ A \rightarrow Ac|Sd|c \end{cases}$

Le non-terminal *S* est récursif à gauche car $S \rightarrow Aa \rightarrow Sda$ (mais il n'est pas immédiatement récursif à gauche).

Eliminitation de la récursivité à gauche pour toute grammaire sans règle $A \rightarrow \varepsilon$

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

Récursivité à gauche non immédiate Exemple

Théorème La grammaire ainsi obtenue reconnait le même langage que la grammaire initiale.

Sur l'exemple : $\begin{cases} S \to Aa|b \\ A \to Ac|Sd|c \end{cases}$

On ordonne S,A

i=1 pas de récursivité immédiate dans $S \to Aa|b$

i=2 et j=1 on obtient $A \to Ac|Aad|bd|c$

on élimine la rec. immédiate : $A \to bdA'|cA'$ $A' \to cA'|adA'|\varepsilon$

Bref, on a obtenu la grammaire $\begin{cases} S \to Aa|b \\ A \to bdA'|A' \\ A' \to cA'|adA'|\varepsilon \end{cases}$

Autre exemple : $\begin{cases} S \to Sa|TSc|d \\ T \to SbT|\varepsilon \end{cases}$

On obtient la grammaire : $\begin{cases} S \to TScS'|dS' \\ S' \to aS'|\varepsilon \\ T \to T'|dS'bTT' \\ T' \to ScS'bTT'|\varepsilon \end{cases}$

Or on a $S \to TScS' \to T'ScS' \to ScS$

Grammaire propre

Définitio *Une grammaire est dite* **propre** *si elle ne contient aucune production* $A \to \varepsilon$

Comment rendre une grammaire propre? En rajoutant une production dans laquelle le A est remplacé par ε, ceci pour chaque A apparaîssant en partie droite d'une production, et pour chaque A d'un $A \to \varepsilon$.

Exemple :

$$\begin{cases} S \to aTb|aU \\ T \to bTaTA|\varepsilon \\ U \to aU|b \end{cases} \text{ devient } \begin{cases} S \to aTb|ab|aU \\ T \to bTaTA|baTA|bTaA|baA \\ U \to aU|b \end{cases}$$

Factorisation à gauche

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

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

Au départ, pour savoir s'il faut choisir $S \to bacdAbd$ ou $S \to bacdBcca$, il faut avoir lu la 5ième lettre du mot (un a ou un c). On ne peut donc pas dès le départ savoir quelle production prendre. Ce qui est incompatible avec une grammaire LL(1). (Remarque : mais pas avec une grammaire LL(5), mais ce n'est pas notre problème.)

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

Exemple :
$$\begin{cases} S \to aEbS|aEbSeB|a \\ E \to bcB|bca \\ B \to ba \end{cases}$$

Factorisée à gauche, cette grammaire devient :
$$\begin{cases} S \to aS'' \\ S'' \to EbSS'|\varepsilon \\ S' \to eB|\varepsilon \\ E \to bcE' \\ E' \to B|a \\ B \to ba \end{cases}$$

Conclusion

Si notre grammaire est LL(1), l'analyse syntaxique peut se faire par l'analyse descendante vue ci-dessus. Mais comment savoir que notre grammaire est LL(1) ?
Etant donnée une grammaire

1. la rendre non ambiguë.
   Il n'y a pas de méthodes. Une grammaire ambiguë est une grammaire qui a été mal concue.
2. éliminer la récursivité à gauche si nécessaire
3. la factoriser à gauche si nécessaire
4. construire la table d'analyse

Il ne reste plus qu'a espérer que ça soit LL(1). Sinon, il faut concevoir une autre méthode pour l'analyse syntaxique .

Exemple : grammaire des expressions arithmétiques avec les opérateurs $+ - / \text{ et } *$

$$E \to E+E|E-E|E*E|E/E|(E)|\text{nb}$$

Mais elle est ambiguë. Pour lever l'ambiguïté, on considère les priorités classiques des opérateurs et on obtient la grammaire non ambiguë :

$$\begin{cases} E \to E+T|E-T|T \\ T \to T*F|T/F|F \\ F \to (E)|\text{nb} \end{cases}$$

Après suppression de la récursivité à gauche, on obtient

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

Inutile de factoriser à gauche.
Cette grammaire est LL(1) (c'est l'exemple que l'on a utilisé tout le temps).

Autre exemple : la grammaire

$$\begin{cases} S \to aTb|\varepsilon \\ T \to cSa|d \end{cases}$$

n'est pas LL(1). Or elle n'est pas récursive à gauche, elle est factorisée à gauche et elle n'est pas ambiguë !

Analyse ascendante

Principe : construire un arbre de dérivation du bas (les feuilles, ie les unités lexicales) vers le haut (la racine, ie l'axiome de départ).

Le modèle général utilisé est le modèle par **décallages-réductions**. C'est à dire que l'on ne s'autorise que deux opérations :

- **décallage (shift)** : décaller d'une lettre le pointeur sur le mot en entrée.
- **réduction (reduce)** : réduire une chaîne (suite consécutive de terminaux et non terminaux à gauche du pointeur sur le mot en entrée et finissant sur ce pointeur) par un non-terminal en utilisant une des règles de production.

Exemple

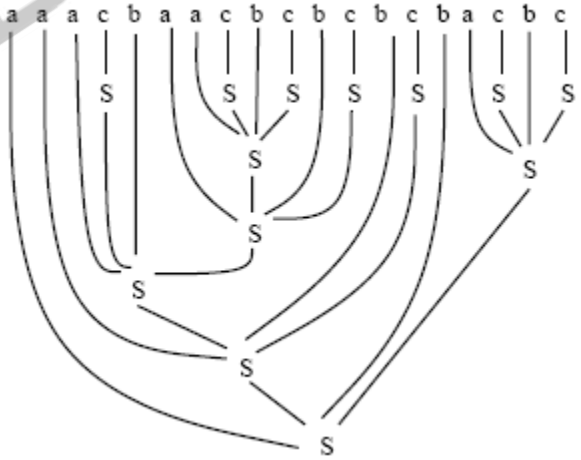
[a] aacbaacbcbcbcbacbc on ne peut rien **réduire**, donc on **décalle**
a [a] acbaacbcbcbcbacbc on ne peut rien réduire, donc on décalle
aa [a] cbaacbcbcbcbacbc on ne peut rien réduire, donc on décalle
aaa [c] baacbcbcbcbacbc ah ! On peut réduire par $S \to c$
aaa [S] baacbcbcbcbacbc on ne peut rien réduire, donc on décalle
...
aaaSbaa [c] bcbcbcbacbc on peut utiliser $S \to c$
aaaSbaa [S] bcbcbcbacbc on ne peut rien réduire, donc on décalle
aaaSbaaS [b] cbcbcbacbc on ne peut rien réduire, donc on décalle
aaaSbaaSb [c] bcbcbacbc On peut utiliser $S \to c$
aaaSbaaSb [S] bcbcbacbc On peut utiliser $S \to aSbS$
aaaSba [S] bcbcbacbc décallage
aaaSbaS [b] cbcbacbc décallage
aaaSbaSb [c] bcbacbc réduction par $S \to c$
aaaSbaSb [S] bcbacbc réduction par $S \to aSbS$
aaaSb [S] bcbacbc réduction par $S \to aSbS$
aa [S] bcbacbc décallage
aaS [b] cbacbc décallage
aaSb [c] bacbc réduction par $S \to c$
aaSb [S] bacbc réduction par $S \to aSbS$
a [S] bacbc décallage
aS [b] acbc décallage
aSb [a] cbc décallage
aSba [c] bc réduction par $S \to c$
aSba [S] bc décallage
aSbaS [b] c décallage
aSbaSb [c] réduction par $S \to c$
aSbaSb [S] réduction par $S \to aSbS$
aSb [S] réduction par $S \to aSbS$
[S] **terminé ! ! ! !** On a gagné, le mot est bien dans le langage.

Soit la grammaire suivante :

S-->aSbS|c

avec le mot

u=aacbaacbcbcbcbacbc



Analyseur LR : description informelle

- La technique utilisée par les analyseurs LR est appelée "*shift-reduce*" (*avance-réduit*) par opposition à "*predict-match*" des analyseurs LL.

- L'idée est de lire les *tokens*, un à un, en le mettant sur la pile. À chaque étape, on analyse la pile pour vérifier si elle contient une sous-chaîne au sommet correspondant à une partie droite d'une production.

  - Si oui, on remplace la sous-chaîne par la partie gauche. Ceci équivaut à une étape de dérivation. Cette étape est appelée "*reduire* (la pile)".
  - Sinon, on continue à lire les tokens, en les déplaçant au sommet de la pile, jusqu'à avoir une sous-chaîne correspondant à la partie droite d'une production. Cette étape est appelée « *shifting (avancer)* (la tête de lecture des *tokens*) » (après avoir déplacer le token courant sur la pile) .

- C'est ça un automate LR...

Exemple

Grammaire

G = (A,V, P,S), avec

A = {*a*, *b*, *c*, *d* , *e*}

V = {*S*, *A*, *B*}

P = {

$S \rightarrow$ aABe,

$A \rightarrow Abc \mid b$

$B \rightarrow d$

}

Entrée: abbcde

	(*pile*	*entrée*)	*action*
1.	($	abbcde$)	Initialization
2.	($a	bbcde$)	Shift ‘a’
3.	($ab	bcde$)	Shift ‘b’
4.	($aA	bcde$)	Reduce ‘b’ ($A \rightarrow b$)
5.	($aAb	cde$)	Shift ‘b’
6.	($aAbc	de$)	Shift ‘c’
7.	($aA	de$)	Reduce ‘Abc’ ($A \rightarrow Abc$)
8.	($aAd	e$)	Shift ‘d’
9.	($aAB	e$)	Reduce ‘d’ ($B \rightarrow d$)
10.	($aABe	$)	Shift ‘e’
11.	($S	$)	Reduce ‘aABe’ ($S \rightarrow$ aABe)
12.	($	$)	Finish : Accept

Table d'analyse LR (1)

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

Construction de la table d'analyse :

- utilise aussi les ensembles SUIVANT (et donc PREMIER), plus ce qu'on appelle des fermetures de 0-items.
- Un 0-item (ou plus simplement *item*) est une production de la grammaire avec un "." quelque part dans la partie droite. Par exemple (sur la gram ETF) :

  E -> E . + T ou encore *T-> F.* ou encore *F -> . (E)*

Table d'analyse LR (2)

Fermeture d'un ensemble d'items I :

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

Exemple : soit la grammaire des expressions arithmétiques

$$\begin{cases} (1) \quad E \to E+T & (3) \quad T \to T*F & (5) \quad F \to (E) \\ (2) \quad E \to T & (4) \quad T \to F & (6) \quad F \to \text{nb} \end{cases}$$

et soit l'ensemble d'items $\{T \to T*.F\ ,\ E \to E.+T\}$.

La fermeture de cet ensemble d'items est : $\{T \to T*.F\ ,\ E \to E.+T\ ,\ F \to .\text{nb}\ ,\ F \to .(E)\}$

Table d'analyse LR (3)

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

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

Sur l'exemple ETF : soit l'ensemble d'items $I = \{T \to T * .F\ ,\ E \to E. + T\ ,\ F \to .\text{nb}\ ,\ F \to .(E)\},$

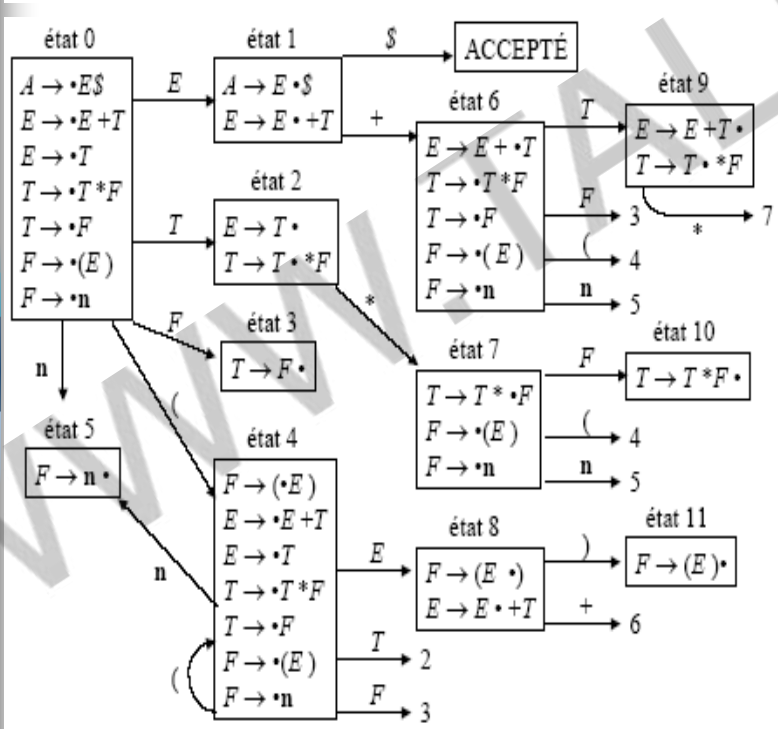
On aura $\Delta(I, F) = \{T \to T * F.\}$

$\Delta(I, +) = \{E \to E + .T\ ,\ T \to .T * F\ ,\ T \to .F\ ,\ F \to .\text{nb}\ ,\ F \to .(E)\}$ etc.

Table d'analyse LR (4)

Collection des items d'une grammaire :

0- Rajouter l'axiome S' avec la production : $S' \rightarrow S$
1- $I_0 \leftarrow$ Fermeture($\{S' \rightarrow .S\}$)
Mettre I_0 dans Collection
2- Pour chaque $I \in$ Collection
Pour chaque X tq $\Delta(I, X)$ est non vide
ajouter $\Delta(I, X)$ dans Collection
finpour
finpour
3- Recommencer 2 jusqu'à ce qu'on n'ajoute rien de nouveau



Toujours sur la grammaire ETF :

$I_0 = \{S \rightarrow .E, E \rightarrow .E+T, E \rightarrow .T, T \rightarrow .T*F, T \rightarrow .F, F \rightarrow .nb, F \rightarrow .(E)\}$
$\Delta(I_0, E) = \{S' \rightarrow E., E \rightarrow E.+T\} = I_1$ (terminal pour la règle $S' \rightarrow E$)
$\Delta(I_0, T) = \{E \rightarrow T., T \rightarrow T.*F\} = I_2$ (terminal pour la règle 2)
$\Delta(I_0, F) = \{T \rightarrow F.\} = I_3$ (terminal pour la règle 4)
$\Delta(I_0, () = \{F \rightarrow (.E), E \rightarrow .E+T, E \rightarrow .T, T \rightarrow .T*F, T \rightarrow .F, F \rightarrow .nb, F \rightarrow .(E)\} = I_4$
$\Delta(I_0, nb) = \{F \rightarrow nb.\} = I_5$ (terminal pour la règle 6)
$\Delta(I_1, +) = \{E \rightarrow E+.T, T \rightarrow .T*F, T \rightarrow .F, F \rightarrow .nb, F \rightarrow .(E)\} = I_6$
$\Delta(I_2, *) = \{T \rightarrow T*.F, F \rightarrow .nb, F \rightarrow .(E)\} = I_7$
$\Delta(I_4, E) = \{F \rightarrow (E.), E \rightarrow E.+T\} = I_8$
$\Delta(I_4, T) = \{E \rightarrow T., T \rightarrow T.*F\} = I_2$ déja vu, ouf
$\Delta(I_4, F) = \{T \rightarrow F.\} = I_3$
$\Delta(I_4, nb) = \{F \rightarrow nb.\} = I_5$
$\Delta(I_4, () = \{F \rightarrow (.E), E \rightarrow .E+T, E \rightarrow .T, T \rightarrow .T*F, T \rightarrow .F, F \rightarrow .nb, F \rightarrow .(E)\} = I_4$
$\Delta(I_6, T) = \{E \rightarrow E+T., T \rightarrow T.*F\} = I_9$ (terminal pour règle 1)
$\Delta(I_6, F) = I_3$
$\Delta(I_6, nb) = I_5$
$\Delta(I_6, () = I_4$
$\Delta(I_7, F) = \{T \rightarrow T*F.\} = I_{10}$ (terminal pour règle 3)
$\Delta(I_7, nb) = I_5$
$\Delta(I_7, () = I_4$
$\Delta(I_8,)) = \{F \rightarrow (E).\} = I_{11}$ (terminal pour règle 5)
$\Delta(I_8, +) = \{E \rightarrow E+.T, T \rightarrow .T*F, T \rightarrow .F, F \rightarrow .nb, F \rightarrow .(E)\} = I_6$
$\Delta(I_9, *) = I_7$

Table d'analyse LR (5)

Construction de la table d'analyse LR :

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

- Si la table construite ne présente aucun conflit (réduction/décalage) pour toute entrée, la grammaire est dite **SLR(1).** Aussi l'analyseur est dit **SLR(1).**
- Toute grammaire ambiguë est non SLR(1), mais toute grammaire non ambiguë n'est pas forcément SLR(1).

Exemple

Toujours sur le même exemple : il nous faut les SUIVANT (et PREMIER) :

$$\begin{cases} (1)\ E \to E+T & (3)\ T \to T*F & (5)\ F \to (E) \\ (2)\ E \to T & (4)\ T \to F & (6)\ F \to \text{nb} \end{cases}$$

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

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

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

Analyseur syntaxique SLR

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

Algorithme :

```
S0 est dans la pile et w$ est dans le tampon d'entrée
Tant que (vrai) faire
   Soit s l'état de sommet de pile
   a le symbole courant pointé par ps
   si M[s,a]=décaler s' alors
                Empiler a puis s'
                Avancer ps
   sinon si M[s,a]=réduire par A-->α alors
                dépiler 2*|α| symboles
                soit s' le nouvel état sommet de Pile
                empiler A et M[s',A]
                émettre en sortie l'identification de la règle
                A--> α
         sinon  si M[s,a]=accepter alors succès()
                sinon erreur():
                finsi
         finsi
   finsi
fintq
```

Exemple

l'analyse du mot 3+*4$:

pile	entrée	action
$ 0	3+*4$	d5
$ 0 3 5	+*4$	r6 : $F \rightarrow$ nb
$ 0 F	+*4$	je suis en 0 avec F : je vais en 3
$ 0 F 3	+*4$	r4 : $T \rightarrow F$
$ 0 T	+*4$	je suis en 0 avec T : je vais en 2
$ 0 T 2	+*4$	r2 : $E \rightarrow T$
$ 0 E	+*4$	je suis en 0 avec E : je vais en 1
$ 0 E 1	+*4$	d6
$ 0 E 1 + 6	*4$	ERREUR!! Ce mot n'appartient pas au langage.

l'analyse du mot 3+4*2$:

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

Remarques

- Cette méthode permet d'analyser plus de grammaires que la méthode descendante (car il y a plus de grammaires SLR que LL)
- On utilise un outil (bison) qui construit tout seul une table d'analyse LR (LALR en fait, mais c'est presque pareil) à partir d'une grammaire donnée
- dans cette méthode d'analyse, ça n'a strictement aucune importance que la grammaire soit récursive à gauche, même au contraire, on **préfère**.

Traitement des erreurs

Emission des messages (diagnostic)

On choisit arbitrairement une des hypothèses possibles

Exemple : **e = a + b c ;**

- opérateur manquant (**e = a + b * c ;**)
- identificateur en trop (**e = a + b ;**)
- erreur lexicale (**e = a + bc ;**)
- ...

Redémarrage

Pour pouvoir compiler la partie après la première erreur

Méthodes de redémarrage

Mode panique

Éliminer les prochains symboles terminaux de la donnée

Mode correction

Modifier les prochains symboles terminaux pour reconstituer ce que serait la donnée sans l'erreur détectée

Le message d'erreur doit être cohérent avec l'hypothèse faite

Règles d'erreur

Ajouter à la grammaire des constructions incorrectes avec leur traitement

La grammaire utilisée par l'analyseur est distincte de celle décrite dans le manuel d'utilisation

Mode panique avec l'analyse LR

X est un non-terminal fixé à l'avance

On suppose que l'erreur est au milieu d'un *X* et on essaie de redémarrer après ce *X*

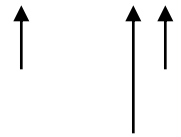
Dépiler jusqu'à un état *q* qui ait une transition par *X*

Eliminer des symboles terminaux de la donnée jusqu'à rencontrer un symbole terminal $a \in$ Suivant(*X*)

Reprendre l'analyse à partir de la case (*q*, *X*) dans la table

Exemple

X = *instr* { *x* = [; *y* = 0 ;



Mode correction avec l'analyse LR

Ajouter dans les cases vides de la table des appels à des fonctions de traitement d'erreur

Les fonctions émettent un message et effectuent des actions pour redémarrer après l'erreur

Exemple

- Les routines d'erreur étant :
- **e1** :
  - (routine appelée depuis les états 0, 2, 4 et 5 lorsque l'on rencontre un opérateur ou la fin de chaîne d'entrée alors qu'on attend un opérande ou une parenthèse ouvrante)
    Emettre le diagnostic operande manquant
    Empiler un nombre quelconque et aller dans l'état 3
- **e2** :
  - (routine appelée depuis les états 0,1,2,4 et 5 à la vue d'une parenthèse fermante)
    Emettre le diagnostic parenthese fermante excedentaire
    Ignorer cette parenthèse fermante
- **e3** :
  - (routine appelée depuis les états 1 ou 6 lorsque l'on rencontre un nombre ou une parenthèse fermante alors que l'on attend un opérateur)
    Emettre le diagnostic operateur manquant
    Empiler + (par exemple) et aller à l'état 4
- **e4** :
  - (routine appelée depuis l'état 6 qui attend un opérateur ou une parenthès fermante lorsque l'on rencontre la fin de chaîne)
    Emettre le diagnostic parenthese fermante oubliee
    Empiler une parenthèse fermante et aller à l'état 9

$$\begin{cases} (1) & E \to E + E \\ (2) & E \to E * E \end{cases} \qquad \begin{matrix} (3) & E \to (E) \\ (4) & E \to \text{nb} \end{matrix}$$

- La table d'analyse LR avec routines d'erreur est

état	nb	+	*	(	)	$	E
0	d3	**e1**	**e1**	d2	**e2**	**e1**	1
1	**e3**	d4	d5	**e3**	**e2**	ACC	
2	d3	**e1**	**e1**	d2	**e2**	**e1**	6
3	r4	r4	r4	r4	r4	r4	
4	d3	**e1**	**e1**	d2	**e2**	**e1**	7
5	d3	**e1**	**e1**	d2	**e2**	**e1**	8
6	**e3**	d4	d5	**e3**	d9	**e4**	
7	r1	r1	d5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

Règles d'erreur

Si l'on a une idée assez précise des erreurs courantes qui peuvent être rencontrées, il est possible d'augmenter la grammaire du langage avec des productions qui engendrent les constructions erronées.

Par exemple (pour un compilateur C) :

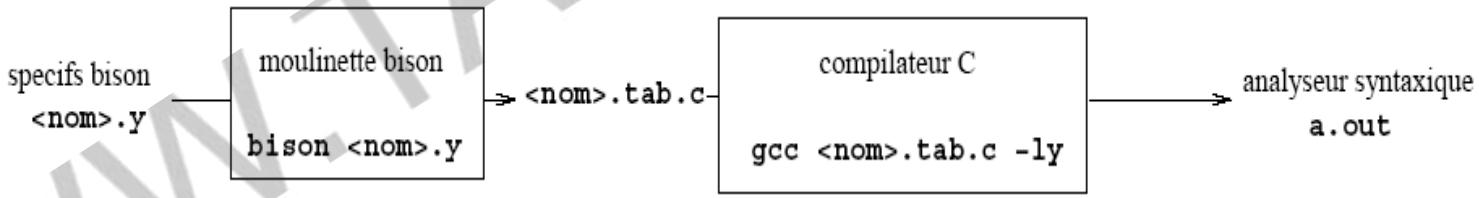
$I \rightarrow \texttt{if}\ E\ I$ (erreur : il manque les parenthèses)

$I \rightarrow \texttt{if}\ (\ E\ )\ \texttt{then}\ I$ (erreur : il n'y a pas de `then` en C)

L'outil yacc/bison

- yacc est un utilitaire d'unix, bison est un produit gnu. yacc/bison accepte en entrée la description d'un langage sous forme de règles de productions et produit un **programme** écrit dans un langage de haut niveau (ici, le langage C) qui, une fois compilé, reconnait des phrases de ce langage (ce programme est donc un analyseur syntaxique).

- yacc signifie *Yet Another Compiler Compiler*, c'est à dire *encore un compilateur de compilateur*. Cela n'est pas tout à fait exact, yacc/bison tout seul ne permet pas d'écrire un compilateur, il faut rajouter une analyse lexicale (à l'aide de (f)lex par exemple) ainsi que des **actions sémantiques** pour l'analyse sémantique et la génération de code.



- yacc/bison construit une table d'analyse LALR qui permet de faire une analyse ascendante. Il utilise donc le modèle **décallages-réductions**.

Structure du fichier de spécifications bison

```
%{
Partie I.1 : déclaration (en C) de variables, constantes, inclusions de fichiers, ...
%}
Partie I.2 : déclaration des unités lexicales utilisées
             déclaration de priorités et de types
%%
Partie II : règles de production avec éventuellement des actions sémantiques
%%
Partie III : bloc principal (fonction main) et fonctions C supplémentaires
```

- Les règles de production (la grammaire)

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

- Les symboles terminaux utilisables dans la description du langage sont
  - des **unités lexicales** que l'on doit impérativement déclarer dans la partie I.2 par %token *nom*. Par exemple :
    %token MC_sinon
    %token NOMBRE
  - des **caractères** entre quotes. Par exemple : '+' 'a'

- Les symboles non-terminaux sont n'importe quelle suite d'une ou plusieurs lettres majuscules et/ou minuscules non déclarées comme unités lexicales. yacc/bison fait la différence entre majuscules et minuscules. SI et si ne désignent pas le même objet.

Structure du fichier de spécifications bison

- La partie III doit contenir une fonction yylex() effectuant l'analyse lexicale du texte, car l'analyseur syntaxique l'appelle chaque fois qu'il a besoin du terminal suivant. Cette fonction doit retourner les unités lexicales et les caractères reconnus. Par exemple elle peut contenir

```
return '*';
return NOMBRE;
```

On peut

- soit écrire cette fonction
- soit utiliser la fonction produite par un compilateur (f)lex. Dans ce cas, il faut :
  - Utiliser le compilateur bison avec l'option –d pour produire le fichier nom.tab.h
  - Inclure ce nom.tab.h au début du fichier de spécifications flex
  - Créer un .o à partir du lex.yy.c produit par flex
  - Lors de la compilation C du fichier nom.tab.c, il faut faire le lien avec ce .o et rajouter la bibliothèque flex lors de la compilation C du fichier nom.tab.c avec : gcc nom.tab.c lex.yy.c –ly –lfl

Structure du fichier de spécifications bison

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

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

- A chaque symbole (terminal ou non) est associé un attribut (i.e une valeur) (de type entier par défaut). L'attribut d'un symbole terminal est la valeur contenue dans la variable globale prédéfinie yylval. Il faudra donc penser a affecter correctement yyval lors de l'analyse lexicale, par exemple :

  [0-9]* {yyval = atoi(yytext); return NOMBRE;}

- Les attributs peuvent être utilisée dans les actions sémantiques (comme un attribut **synthétisé**). Le symbole $$ référence la valeur de l'attribut associé au non-terminal de la partie gauche, tandis que $i référence la valeur associée au i-ème symbole (terminal ou non-terminal) ou **action sémantique** de la partie droite. Exemple :
  expr : expr '+' expr { tmp=$1+$3;} '+' expr { $$=tmp+$6;};
  Par défaut, lorsqu'aucune action n'est indiquée, yacc/bison génère l'action {$$=$1;}

Communication avec l'analyseur lexical : yylval

- La variable yylval est de type int par défaut. On peut changer ce type par la déclaration dans la partie I.2 d'une union, par exemple :

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

- Avec cet exemple on pourra stocker dans yylval aussi bien des entiers que des réels ou encore des chaines. Dans ce cas, il faut typer les unités lexicales dont on utilise l'attribut, en utilisant les champs de l'union :

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

ainsi que les symboles non-terminaux dont on utilise l'attribut :

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

- Et attention, puisque la variable est maintenant une union, il faut préciser son champ quand on l'utilise par exemple dans l'analyse lexical on peut rencontrer :

```
{nombre}  {yyval.entier=atoi(yytext); return NB;}
```

Variables, fonctions et actions prédéfinies

yyparse() :	fonction qui lance l'analyseur syntaxique.
YYACCEPT :	instruction qui permet de stopper l'analyseur syntaxique.
int main() :	le main par défaut se contente d'appeler yyparse(). L'utilisateur peut écrire son propre main dans la partie III.
yyerror(char *) :	fonction appelée chaque fois que l'analyseur est confronté à une erreur. Cette fonction se contente d'afficher le message parse error. On peut la redéfinir dans la partie III.
%start *non-terminal* :	action pour signifier quel non-terminal est l'axiome. Par défaut, c'est le premier décrit dans les règles de production.

Conflits shift-reduce et reduce-reduce

- Lorsque l'analyseur est confronté à des conflits, il rend compte du type et du nombre de conflits rencontrés

  >bison exemple.y

  conflicts : 6 shift/reduce, 2 reduce/reduce

  Bison résoud les conflits de la manière suivante (voir égalemant le fichier non.output) :
  - Conflit reduce/reduce : la production choisie est celle apparaissant en premier dans la spécification.
  - Conflit shift/reduce : c'est le shift qui est effectué.

- On peut modifier cette façon de résoudre les conflits en donnant des associativités (droite ou gauche) et des priorités aux symboles terminaux.
  Les déclarations suivantes (dans la partie I.2)

  %left *term1 term2*
  %right *term3*
  %left *term4*
  %nonassoc *term5*

  indiquent que les symboles terminaux *term1*, *term2* et *term4* sont associatifs à gauche, *term3* est associatif à droite, alors que *term5* n'est pas associatif. Les priorités des symboles sont données par l'ordre dans lequel apparait leur déclaration d'associativité, les premiers ayant la plus faible priorité. Lorsque les symboles sont dans la même déclaration d'associativité, ils ont la même priorité.

- La priorité (ainsi que l'associativité) d'une production est définie comme étant celle de son terminal le plus à droite. Mais si l'on fait suivre la production de la déclaration :

  %prec *terminal-ou-unite-lexicale*

  Cela a pour effet de donner à la production la priorité et l'associativité du *terminal-ou-unite-lexicale.*

- Du coup, un conflit shift/reduce, i.e. un choix entre une réduction A-->α et un décalage d'un symbole d'entrée *a*, est alors résolu en appliquant les règles suivantes :

  – si la priorité de la production A-->α est supérieure à celle de *a*, c'est la réduction qui est effectuée
  – si les priorités sont les mêmes et si la production est associative à gauche, c'est la réduction qui est effectuée
  – dans les autres cas, c'est le shift qui est effectué.

Récupération des erreurs

- Lorsque l'analyseur produit par bison rencontre une erreur, il appelle par défaut la fonction yyerror(char *) qui se contente d'afficher le message parse error, puis il s'arrête. Cette fonction peut être redéfinie par l'utilisateur.

- Il est possible de traiter de manière plus explicite les erreurs en utilisant le mot clé bison error. On peut rajouter dans toute production de la forme

  A--> α1|α2|… une production ***A--> error β***

  Dans ce cas, Dès qu'une erreur est rencontrée, tous les caractères sont avalés jusqu'à rencontrer le caractère correspondant à ***β*** .

  Exemple : ***La production instr--> error ';'***

  indique à l'analyseur qu'à la vue d'une erreur, il doit sauter jusqu'au delà du prochain ";" et supposer qu'une *instr* vient d'être reconnue.

  La routine yyerrok replace l'analyseur dans le mode normal de fonctionnement c'est à dire que l'analyse syntaxique n'est pas interrompue.

Exemples de fichier .y

Cet exemple reconnait les mots de la forme $wc\tilde{w}$ où w est un mot sur {a,b}. Cet exemple ne fait pas appel à la fonction yylex générée par le compilateur (f)lex.

miroir.y

```
%%
mot : S '$' {printf("mot accepte\n");YYACCEPT;}
    ;

S : 'a' S 'a'
  | 'b' S 'b'
  | 'c'
  ;

%%
int yylex() {
  char c=getchar();
  if (c=='a' || c=='b' || c=='c' || c=='$') return(c);
  else printf("ERREUR : caractere non reconnu : %c ",c);
}
```

Exemples de fichier .y

Ce second exemple reconnait des listes d'entiers séparés par des virgules, se terminant par un point, et précédées soit du mot somme, soit du mot produit. A chaque liste précédée du mot somme (resp, du mot produit), l'exécutable affiche la somme (resp, le produit) des entiers formant la liste.

Cet exemple utilise la fonction yylex générée par le compilateur flex.

Le fichier de spécifications flex : listes.l (analyseur lexical)

fichier de spécifications pour flex : listes.l (analyseur lexical)

```
%{
#include <stdlib.h>
#include "listes.tab.h"  // INCLUSION DES DEFS DES TOKEN
%}
%%
somme       return(SOM);
produit     return(PROD);
[.,]        return(yytext[0]);
[0-9]+      {
              yylval=atoi(yytext);
              return(NOMBRE);
            }
[ \t\n]     /* rien */;
"$"         return(FIN);
<<EOF>>     return(FIN);
.           printf("ERREUR : %c inconnu\n", yytext[0]);
```

Exemples de fichier .y

Le fichier de spécifications bison : listes.y (analyseur syntaxique)

```
%{
int res;
%}
%token SOM PROD
%token NOMBRE
%token FIN
%%
liste : FIN {printf("Merci et a bientot\n"); YYACCEPT;}
      | SOM listesom '.' {printf("the somme is %d\n",res);} liste
      | PROD listeprod '.' {printf("the produit is %d\n",res);} liste
      ;

listesom : NOMBRE {res=yylval;}
         | listesom ',' NOMBRE {res+=yylval;}
         ;

listeprod : NOMBRE {res=yylval;}
          | listeprod ',' NOMBRE {res*=yylval;}
          ;
```

Et un Makefile pour relier tout ça :

```
EXO : listes
$(EXO) : lex.yy.o $(EXO).tab.c
	gcc $(EXO).tab.c lex.yy.o -o $(EXO) -ly -lfl
lex.yy.o : lex.yy.c
	gcc -c lex.yy.c
lex.yy.c : $(EXO).l $(EXO).tab.h
	flex $(EXO).l
$(EXO).tab.h : $(EXO).y
	bison -d $(EXO).y
$(EXO).tab.c : $(EXO).y
	bison $(EXO).y
```