Algorithmique et bases de la programmation...

Introduction à la Compilation

Nicolas Delestre

INSA ROUEN

asi Architecture des Systèmes d'Information

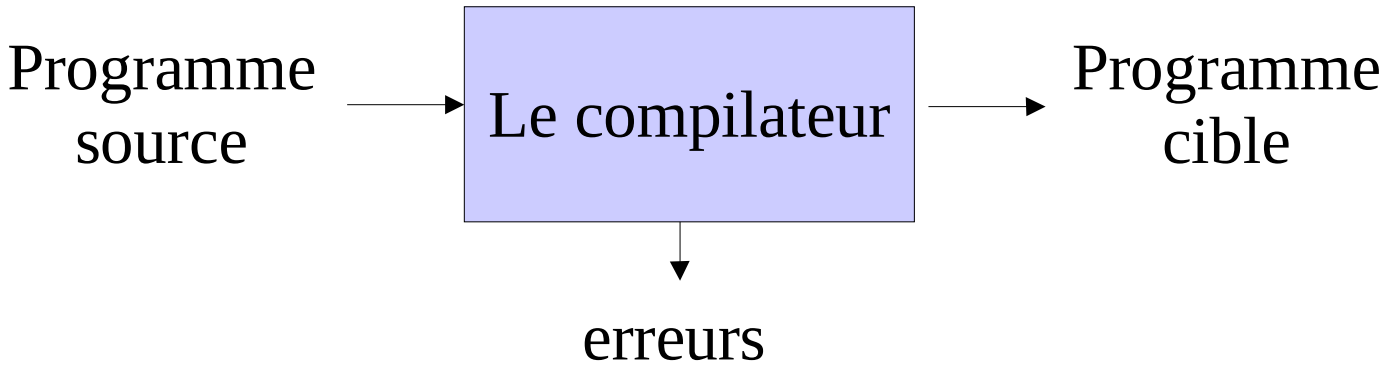
Copyright...

Note de l'auteur...

- Ce document est une **introduction** à la compilation
- Il n'est en aucun cas un cours approfondi décrivant les différentes étapes d'un compilateur

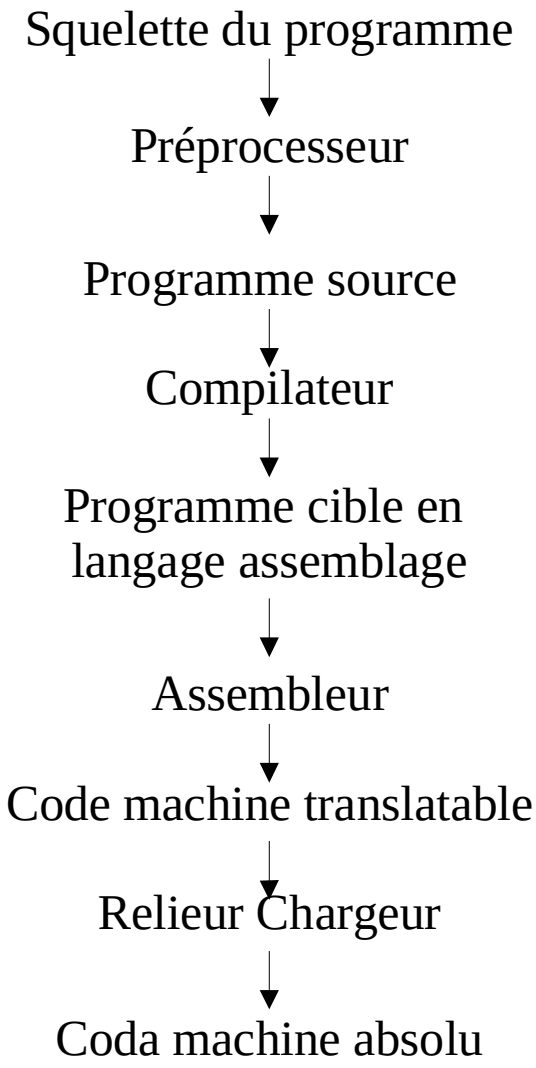
Généralités...

- Qu'est ce qu'un compilateur ?
  - C'est un programme qui traduit un programme écrit dans un langage source vers un langage cible en indiquant les erreurs éventuelles que pourrait contenir le programme source
  - En général le langage cible est du code machine



Généralités...

- Les différentes étapes :
  - De votre programme source...
  - ...jusqu'au programme exécutable



Préprocesseurs...

- Produire ce qui sera le texte d'entrée d'un compilateur
- Macro-expansion
  - définir et utiliser des macros (abréviations)

```
#define max(A, B) ((A) > (B) ? (A) : (B))
...
x = max(p+q, r+s);
```

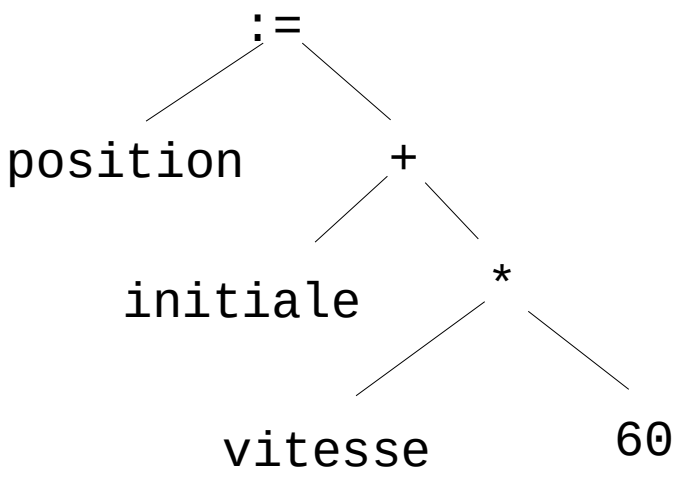
→ x = ((p+q) > (r+s) ? (p+q) : (r+s));

Préprocesseurs...

- Inclusion de fichiers d'en-tête
  - #include <stdio.h>
- Préprocesseurs "rationnels"
  - Actualiser les structures de données et de contrôle d'un langage ancien
    - tantque en FORTRAN
- Extensions de langages
  - Etendre les fonctionnalités d'un langage par des macros intrinsèques

Modèle de compilation par Analyse et Synthèse...

- Exemple : *position := initiale + vitesse * 60*
- Partie analyse :
  - Partition du programme source
  - Représentation intermédiaire : **arbre abstrait**
- Partie synthèse :
  - Programme cible à partir d'un arbre abstrait
    - Notations :
      - F : flottants
      - # : constantes



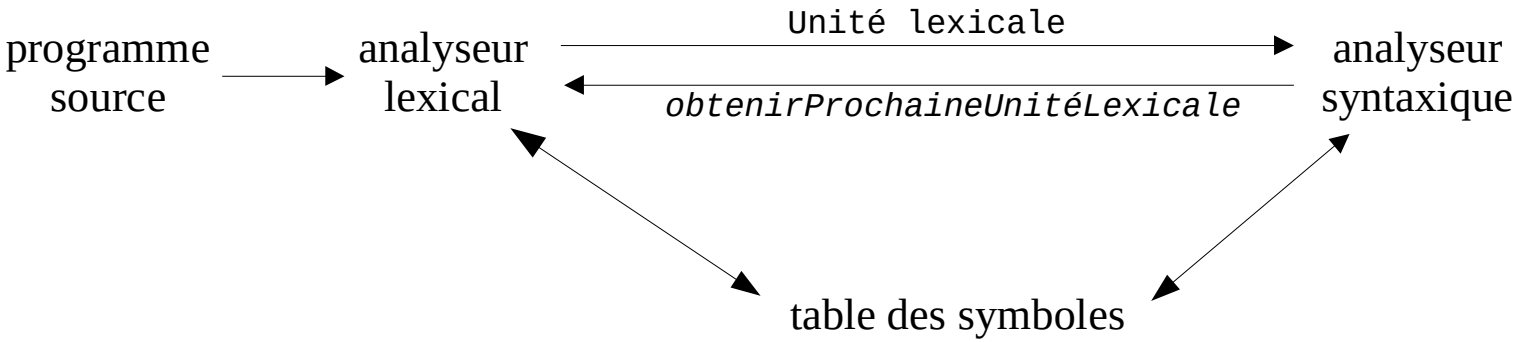

```
MOVF    id3, R2
MULF    #60.0, R2
MOVF    id2, R1
ADDF    R2, R1
MOVF    R1, id1
```

Analyse lexicale...

- But :
  - Flot de caractères $\rightarrow$ flot d'unités lexicales
- Vocabulaire :
  - Lexème : Suite de caractères ayant un sens (position, :=, initiale, ...)
  - Unité lexicale : Nom d'une classe de lexèmes (identifiant, nombre, …)
  - Valeur lexicale : Valeur attachée à certaines unités lexicales

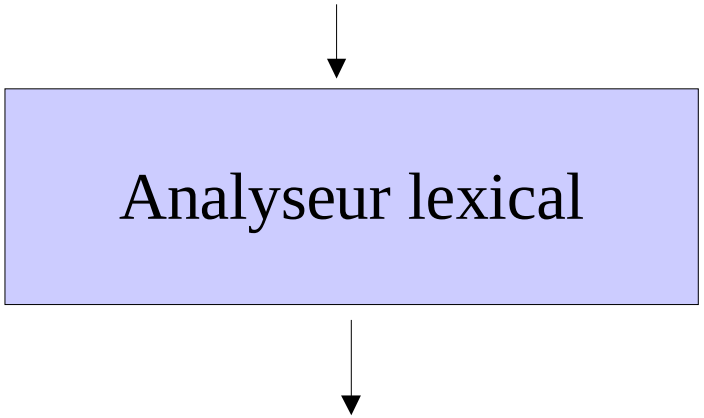
Rôle de l'analyseur lexical...

- Lire les caractères du texte d'entrée
- Supprimer les blancs, les commentaires, etc.
- Former des lexèmes
- Passer des couples <unités lexicales,valeur lexicale> à l'Analyseur Syntaxique



Exemple...

```
position := initiale + vitesse * 60
```



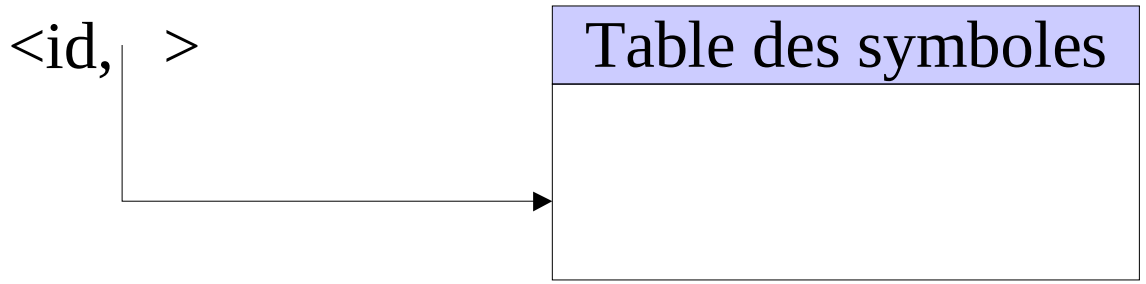

```
id1  :=  id2 +  id3  *  60
```

Traitement...

- ...des "blancs" et des commentaires :
  - Comment ? Modifier la grammaire, difficile...
  - Que faire ?
    - Supprimer les blancs : espaces, tabulations, fins de lignes,
      - dépend du langage
    - Supprimer les commentaires
- ...des constantes (nombres)
  - Comment ? Ajouter des productions à la grammaire...
  - Que faire ?
    - Regrouper les chiffres en nombre
    - Passer nb à l'analyseur syntaxique avec sa valeur

Traitement...

- ...des identificateurs
  - Exemple :
    - `compte = compte + incrément` $\Rightarrow$ `id = id + id`
  - Utilisation d'une unité lexicale particulière : *id*
  - La valeur d'un *id* est son rang dans une *table des symboles*



Traitement...

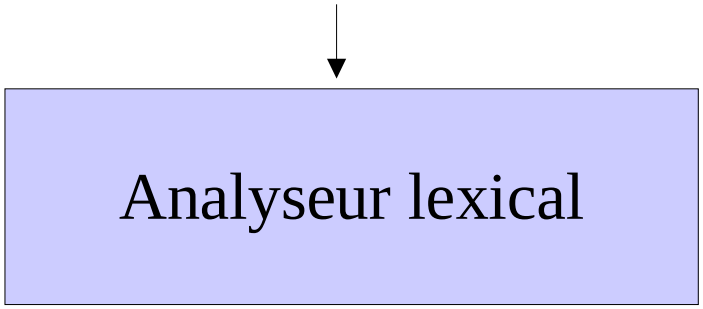
- ...des mots clefs :
  - *begin, end, if, …*
  - Dans la plupart des cas ces mots sont réservés
    - Ils sont déjà présents dans la table des symboles et ne peuvent pas être modifiées
  - ... mais il existe des langages qui permettent de redéfinir ces mots clefs

Table des symboles...

- Utilisée pour ranger des informations sur les diverses constructions du langage source
- Les informations sont collectées par les phases d'analyse du compilateur et utilisées par les phases de synthèse pour engendrer le code objet
  - Nom du lexème, type, utilisation, adresse mémoire
- Possède deux fonctions :
  - *Insérer(Chaîne, Unité lexicale)*
  - *Chercher(Chaîne)*

Exemple...

```
position := initiale + vitesse * 60
```




```
<id,4>
<op_affectation, >
<id,5>
<op_addition,>
<id,10>
<op_multiplication,>
<nb,60>
```

Spécification des unités lexicales...

- Les chaînes
- Les langages
- Les expressions régulières
- Les définitions régulières

Les chaînes...

- Les chaînes (ou *mot* et *phrase*):
  - Alphabet = ensemble fini de symboles
  - Une chaîne sur un alphabet est un ensemble fini de symboles
  - ε représente la chaîne vide
  - Si s est une chaîne, |s| représente sa taille (| ε|=0)
  - Opérations sur les chaînes :
    - ***ban*** est un préfixe de ***ban****ane*
    - ***ane*** est un suffixe de *ban****ane***
    - ***ana*** est une sous-chaîne de *b****ana****ne*
    - ***baan*** est une sous-suite de ***ba****n****an****e*

Les chaînes...

- Fonctions sur les chaînes :
  - Concaténation :
    - Si s et t sont deux chaînes, st est la concaténation de s et t
    - $\varepsilon s = s\varepsilon = s$
  - Exponentiation :
    - $s^n = s^{n-1}s$ (pour $n>0$) et $s^0 = \varepsilon$

Les langages...

- Les langages :
  - Langage = ensemble quelconque de chaînes construites pour un alphabet fixé
  - Ø est l'ensemble vide
  - {ε} est l'ensemble ne contenant que la chaîne vide
  - Opérations sur les langages, soit L et M deux langages:
    - $L \cup M = \{s \mid s \in L \text{ ou } s \in M\}$
    - $LM = \{st \mid s \in L \text{ ou } t \in M\}$
    - $L^* = \cup^{\infty}_{i=0} L^i$ fermeture de *Kleene*
    - $L^+ = \cup^{\infty}_{i=1} L^i$ fermeture positive de *Kleene*

Les expressions régulières...

- Expressions régulières (rationnelles) :
  - Notation permettant d'identifier les éléments d'un langage
    - Par exemple, *lettre (lettre* | *chiffre)** est la "forme" des identificateurs en Pascal
  - On construit une expression régulière à partir d'expressions régulières plus simples en utilisant un ensemble de règles de définition
  - Chaque expression *r* dénote un langage noté *L(r)*

Les expressions régulières...

- Les règles qui définissent les expressions régulières sur un alphabet $\sum$ sont :
  - l'expression régulière ε, qui dénote le langage $\{\varepsilon\}$
  - Si $a \in \sum$ alors l'expression régulière alors *a* dénote le langage {a}
  - Si *r* et *s* sont des expressions régulières dénotant *L(r)* et *L(s)* :
    - *(r)* | *(s)* dénote $L(r) \cup L(s)$
    - *(r) (s)* dénote *L(r) L(s)*
    - $(r)^*$ dénote $(L(r))^*$

Les expressions régulières...

- Un langage dénoté par une expression régulière est un **ensemble régulier**
- On peut enlever les () superflues si on adopte les conventions suivantes :
  - l'opérateur unaire * possède la plus haute priorité et est associatif à gauche
  - la concaténation possède la 2ème priorité et est associative à gauche
  - | possède la plus basse priorité et est associative à gauche
  - … donc (a)|((b)*(c)) est équivalent à a|b*c

Les expressions régulières...

- Deux expression régulières *r* et *s* sont équivalentes si et seulement si les langages qu'elles dénotent sont égaux :
  - $r = s \Leftrightarrow L(r) = L(s)$
- Lois algébriques :
  - $r|s = s|r$
  - $r|(s|t) = (r|s)|t$
  - $(rs)\ t = r(st)$
  - $r(s|t) = rs|rt$ et $(s|t)r = sr|tr$
  - $\varepsilon r = r$ et $r\varepsilon = r$
  - $r^* = (r|\varepsilon)^*$
  - $r^{**} = r^*$

Les expressions régulières...

- Exemples avec $\sum$ = {a,b} :
  - a|b dénote {a,b}
  - (a|b)(a|b) dénote {aa,ab,ba,bb}
  - $(a|b)^*$ dénote des chaînes composées de a et de b
  - $a|a^*b$ dénote {a, b, ab, aab, aaab, ...}
- Certains langages ne peuvent être représentés à l'aide d'expression régulières, on parle alors d'ensembles non réguliers
  - Par exemple :
    - ensemble équilibrés ou imbriqués
    - {wcw | w est une chaîne de a et de b}

Définition régulière...

- Pour des raisons de commodités, on peut souhaiter nommer les expressions régulières
- Une définition régulière est une suite de définitions de la forme :

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\ldots$$
$$d_n \rightarrow r_n$$

où d_i est un nom distinct et r_i est une expression régulière sur $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

Définition régulière...

- Exemple :
  - Définition régulière des identificateurs :

lettre	$\rightarrow$	A|B|C|…|Z|a|b|…|z
chiffre	$\rightarrow$	1|2|3|…|9
id	$\rightarrow$	**lettre**(**lettre**|**chiffre**)*

Notation abrégée...

- $r^+ = rr^*$ et dénote $L(r)^+$
  - donc $r^*=r+|\varepsilon$
  - + et * ont la même priorité
- $r?=r|\varepsilon$

Exemple...

- Nombres non signés en Pascal :
  - chiffre $\rightarrow 0$ | ... | 9
  - chiffres $\rightarrow$ chiffre+
  - fract_opt $\rightarrow$ (.chiffres)?
  - exposant_opt $\rightarrow$ (E(+ | -)? chiffres)?
  - nb $\rightarrow$ chiffres fract_opt exposant_opt

Création d'un analyseur lexical...

- Étapes :
  - Reconnaissance des unités lexicales
    - On définit les définitions régulières du langage
    - On établie une table de correspondance expression régulière - unité lexicale
    - On construit des diagrammes de transition pour chaque unité
  - Concaténation des diagrammes de transition

Exemple...

- Les définitions régulières :
  - **si** $\rightarrow$ si
  - **alors** $\rightarrow$ alors
  - **sinon** $\rightarrow$ sinon
  - **oprel** $\rightarrow$ < | <= | = | <> | => | >
  - **id** $\rightarrow$ lettre (lettre | chiffre)*
  - **nb** $\rightarrow$ chiffre+ (.chiffre+)? (E(+ | -) chiffre+)?

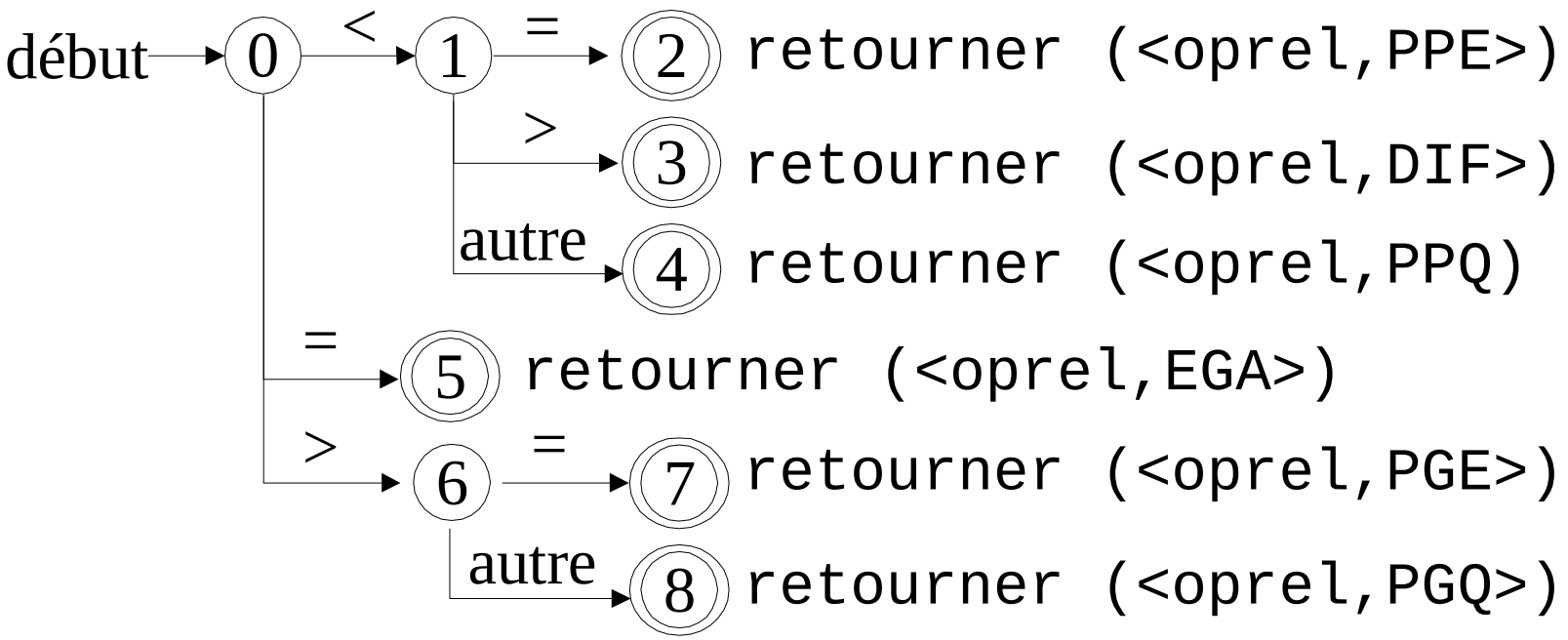
Exemple...

- Le tableau de correspondance expression régulière - unité lexicale

Expression régulière	Unité lexicale	Valeur d'attribut
si	si	**-**
alors	alors	**-**
sinon	sinon	**-**
id	id	ptr vers une entrée
nb	nb	ptr vers une entrée
<	oprel	PPQ
<=	oprel	PPE
=	oprel	EGA
<>	oprel	DIF
>	oprel	PGQ
>=	oprel	PGE

Un exemple...

- Un des diagrammes de transition (celui de l'unité lexicale *oprel*) :



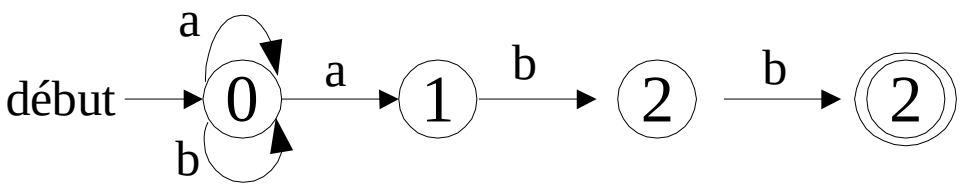
Quelques mots sur les automates...

- Diagrammes de transition issus de la théorie des automates
- Rappels :
  - Un automate est un graphe orienté valué possédant un sommet initial et un ou plusieurs sommets terminaux
  - Deux types :
    - Automates Finis Non déterministe (AFN)
    - Automates Finis Déterministes (AFD)

Automate	Place	Temps	
AFN	O(|r|)	O(|r|*|x|)	x : chaîne de caractères
AFD	O(2|r|)	O(|x|)	r : expression régulière

Quelques mots sur les automates...

- AFN :
  - <E, Σ, δ, e_0, F>
    - E ensemble d'état
    - Σ Alphabet des symboles d'entrée
    - δ une fonction de transition
    - e_0 qui désigne 'état initial
    - F qui désignent les états finaux
  - Représentation graphique
    - Exemple :
      - $(a|b)^*abb$

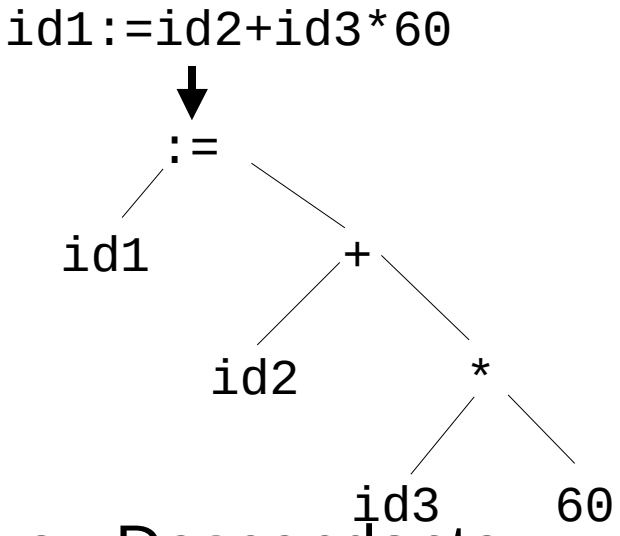


Quelques mots sur les automates...

- AFD :
  - AFD = AFN
    + aucun état a de ε-transition
    + au plus un arc étiqueté par *a* qui sort de *e*
  - Au plus un chemin depuis l'état de départ étiqueté par x

Analyse Syntaxique...

- But :
  - Flot d'unités lexicales → **arbre abstrait**
  - Fonctionnement basé sur les structures hiérarchique d'un programme
  - Utilisation de grammaire non contextuelles (ou notation BNF) pour décrire la syntaxe du langage
  - Deux types d'analyses : Ascendante ou Descendante



Grammaire...

- Grammaire non contextuelle :
  - Permet d'exprimer des règles du type :

    Si S1 et S2 sont des instructions et E une expression, alors

    « si E alors S1 sinon S2 » est une instruction
  - ….de la façon suivante :

    instr → **si** *expr* **alors** *instr* **sinon** *instr*
  - Composée :
    - symboles terminaux (les unités lexicales)
    - symboles non-terminaux (gauche des règles)
    - l'axiome (non-terminal particulier, de départ)
    - les productions (les règles)

Grammaire...

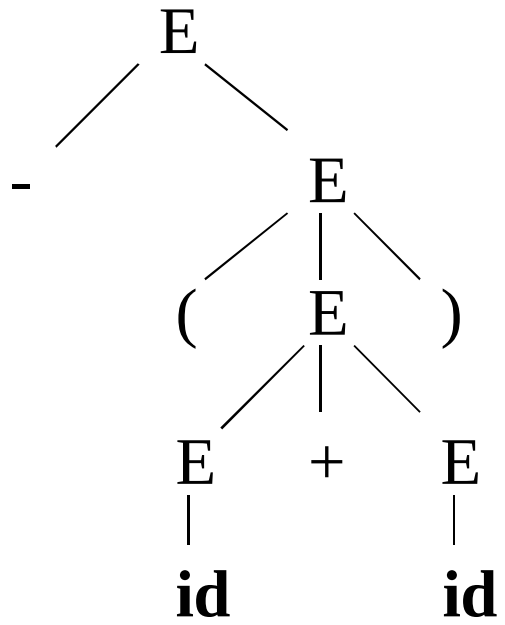
- les conventions de notation :
  - Terminaux : 'a'..'z', '+','*'…,'(','.',…,'0'..'9', **chaîne en gras**
  - Non-terminaux : lettres majuscules du début de l'alphabet, *noms minuscules en italique*
  - Axiome : S
  - Symboles grammaticaux : lettres majuscules de fin de l'alphabet
  - Chaîne de symboles grammaticaux : ‘ α ’, ’β ’,…
  - Utilisation du symbole |
  - Exemple :
    - E $\rightarrow$ E A E | E | (E) | -E | **id**
    - A $\rightarrow$ + | - | * | / | ^

Dérivation...

- Chaîne obtenue en commençant par l'axiome et en remplaçant de manière répétée un non-terminal par la partie droite d'une des productions le définissant
  - $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$
  - Notation :
    - $\alpha A \beta \Rightarrow \alpha \gamma \beta$ (Si il existe $A \rightarrow \gamma$)
    - Dérivation en 0 ou plusieurs étapes : $\Rightarrow^*$
    - Dérivation en 1 ou plusieurs étapes : $\Rightarrow^+$
    - Dérivation gauche : $\Rightarrow_g$
    - Dérivation droite : $\Rightarrow_d$

Dérivation...

- Une dérivation peut alors donner un arbre d'analyse :
  - E ⇒ -E ⇒ -(E) ⇒ -(E+E) ⇒ -(**id**+E) ⇒ -(**id**+ **id**)



Remarques...

- Chaque construction qui peut être décrite par un expression régulière peut également être décrite par une grammaire
- Pourquoi utiliser les expressions régulières pour définir la syntaxe de la partie lexicale d'un langage?
  - Les règles lexicales sont en général plus simple, on n'a pas besoin de la puissance des grammaires
  - Les expressions régulières constituent en général une notation plus concise et plus facile à appréhender pour les unités lexicales que les grammaires

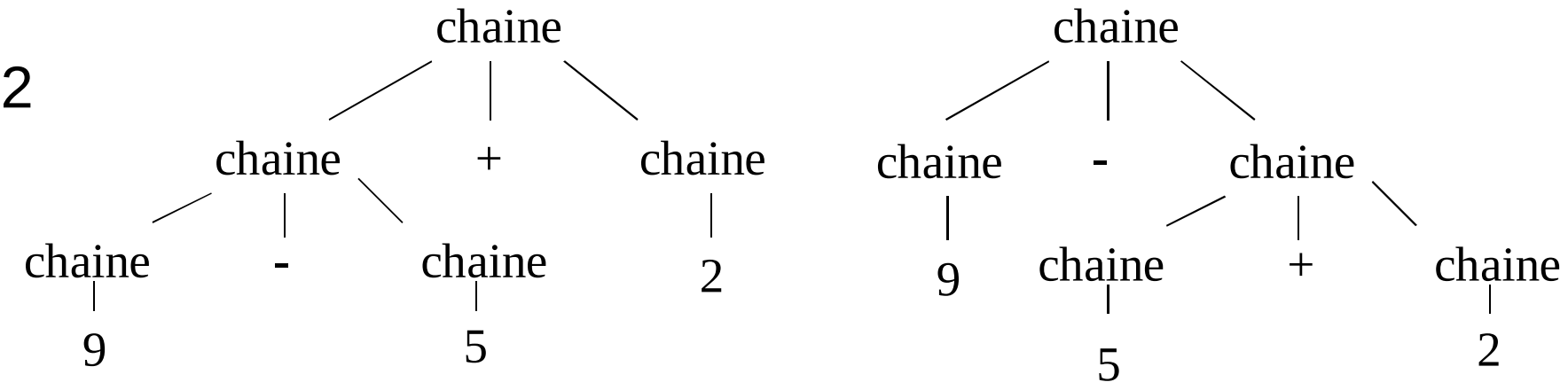
Remarques...

- Des analyseurs lexicaux plus efficaces peuvent être construits automatiquement à partir d'expressions régulières
- Séparer la structure syntaxique des langages en une partie lexicale et une partie non lexicale permet de modulariser la partie frontale d'un compilateur en deux composants de taille raisonnable

Grammaire ambiguë...

- Un grammaire est dit ambiguë lorsqu'au moins une chaîne peut être engendrée par des arbres d'analyse différents
- Exemples :
  - chaine $\rightarrow$ chaine+chaine | chaine-chaine | 0 |..| 9 |

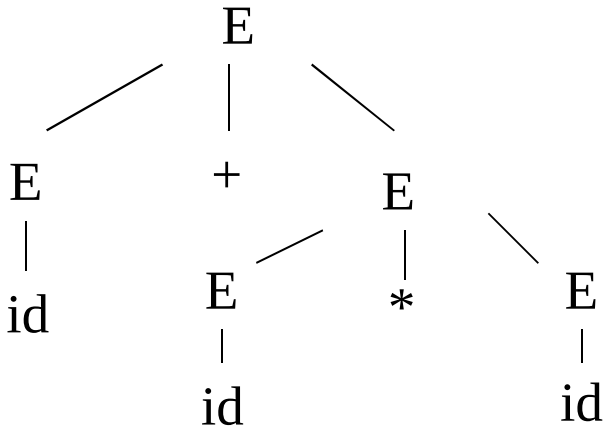
9-5+2



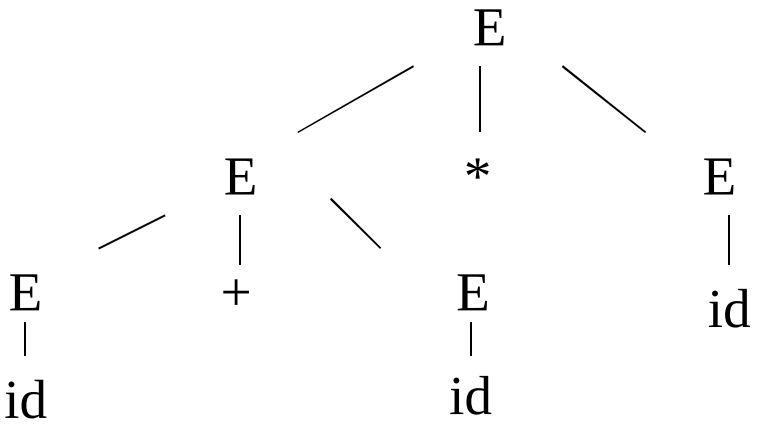
Grammaire ambiguë...

- E $\rightarrow$ E+E | E-E | E*E | E/E | (E) | -E | **id**
  Deux dérivation sont possibles pour obtenir **id+id*id**

$$
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow \mathbf{id} + E \\
&\Rightarrow \mathbf{id} + E * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
$$



$$
\begin{aligned}
E &\Rightarrow E * E \\
&\Rightarrow E + E * E \\
&\Rightarrow \mathbf{id} + E * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
$$



Suppression des ambiguïtés...

- Modifier la grammaire :
  - E $\rightarrow$ E +T | E-T | T
  - T $\rightarrow$ T * F | T / F | F
  - F $\rightarrow$ (E) | -E | **id**
- Il n'y a pas de technique automatique....

Suppression des ambiguïtés...

- Associativité des opérateurs :
  - Exemple :
    - Comment comprendre : 9+5+2 ou a=b=2 ?
  - Associativité à gauche :
    - Un opérande entouré par un même opérateur sera traité par celui de gauche (c'est le cas de +,-,*,/)
  - Associativité à droite :
    - Un opérande entouré par un même opérateur sera traité par celui de droite (c'est le cas de l'affectation)
- Priorité des opérateurs :
  - * à une priorité supérieure à + signifie que * agit sur ses opérandes avant +

Récursivité à gauche...

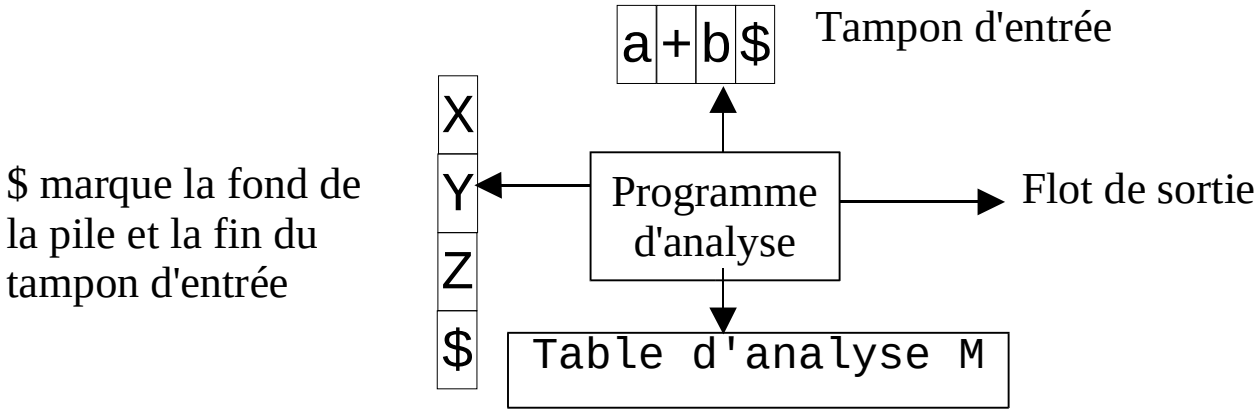
- Une grammaire est récursive à gauche si elle contient un non terminal A tel qu'il existe une dérivation $A \Rightarrow^{+} A\alpha$
  - Une grammaire récursive à gauche ne peut pas être utilisée par un analyseur descendant
  - Élimination de la récursivité à gauche
    - Si on a la règle : $A \rightarrow A\alpha \mid \beta$
    - On la remplace par : $A \rightarrow \beta R$ et $R \rightarrow \alpha R \mid \varepsilon$

Récursivité à gauche...

- Exemple :
  - $E \rightarrow E + T \mid E - T \mid T$
  - $T \rightarrow T * F \mid T / F \mid F$
  - $F \rightarrow (E) \mid -E \mid \mathbf{id}$
- La même grammaire dérécursivée :
  - $E \rightarrow TE'$
  - $T \rightarrow FT'$
  - $F \rightarrow (E) \mid -E \mid \mathbf{id}$
  - $E' \rightarrow +TE' \mid -TE' \mid \varepsilon$
  - $T' \rightarrow *FT' \mid /FT' \mid \varepsilon$

Analyse descendante...

- Il est possible de construire un analyseur prédictif non récursif, il utilise :
  - Le tampon d'entrée (issu de l'analyse lexicale)
  - Une pile de non terminaux
  - Une table d'analyse, qui pour un non terminal donnée et un ou plusieurs terminaux données détermine le ou les règles qui peuvent être utiliser



Analyse descendante...

- On construit la table d'analyse à l'aide de :
  - *Premier(α)* désigne l'ensemble des terminaux qui commencent les chaînes qui se dérivent de α
  - *Suivant(A)* définit l'ensemble des terminaux qui peuvent apparaître immédiatement à droite de A

Premier(X)...

- Algorithme *grossier* :
  1. Si X est un terminal, Premier(X)={X} et Premier(ε)={ε}
  2. Si A→bα alors {b}∈Premier(A)
  3. Si A→Bα alors Premier(B)⊂Premier(A)
  4. De plus si A→Bα et B⇒*ε et a=Xβ alors Premier(X)⊂Premier(A) et réitérer l'opération 3 pour Ba=Xb
- Exemple :

E → TE ’ ⟶ Premier(T)⊂ Premier(E) donc Premier(E)={(,id}
E ’ → +TE ’ | ε ⟶ Premier(E ’)={+, ε}
T → FT ’ ⟶ Premier(F)⊂ Premier(T) donc Premier(T)={(,id}
T ’ → *FT ’ | ε ⟶ Premier(T ’)={*, ε}
F → (E) | **id** ⟶ Premier(F)= {(,id}

Suivant(X)...

- Algorithme *grossier* :
  1. Mettre $ dans Suivant(S) ou S est l'axiome et $ la marqueur de fin du texte source
  2. S'il existe une production $A \rightarrow \alpha B \beta$, le contenu de Premier(β), excepté ε, est ajouté à Suivant(B)
  3. S'il existe une production $A \rightarrow \alpha B$ ou une production $A \rightarrow \alpha B \beta$ telle que Premier(β) contient ε, les éléments de Suivant(A) sont ajoutés à Suivant(B)
- Exemple :

$E \rightarrow TE' \longrightarrow \{\$\} \in$ Suivant(E) et Suivant(E)$\subset$Suivant(E')

$E' \rightarrow +TE' \mid \varepsilon \longrightarrow$ Premier(E')$\subset$Suivant(T) et Suivant(E')$\subset$Suivant(T)

$T \rightarrow FT' \longrightarrow$ Suivant(T)$\subset$Suivant(T') et Premier(T')$\subset$Suivant(F)

$T' \rightarrow *FT' \mid \varepsilon \longrightarrow$ Suivant(T')$\subset$Suivant(F)

$F \rightarrow (E) \mid \mathbf{id} \longrightarrow \{)\} \in$ Suivant(E)

Suivant(X)...

- Donc :

```
Suivant(E)      =Suivant(E')={),$}
Suivant(T)      =Suivant(T')={+,),$}
Suivant(F)      ={+,*,),$}
```

Construction de la table d'analyse...

- Algorithme *grossier* :
  1. Pour chaque production A→α ,procéder aux étapes 2 et 3
  2. Pour chaque terminal a dans Premier(α), ajouter A→α à M[A,a]
  3. Si ε est dans Premier(α), ajouter A→α à M[A,b] pour chaque terminal b dans Suivant(A). Si ε est dans Premier(α) et $ est dans Suivant(A), ajouter A→α à M[A,$]
  4. Faire de chaque entrée non définie de M une erreur
- Exemple :

Non-terminal	Symbole d'entrée					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	E→id			F→(E)		

Grammaire LL(1)...

- Signification :
  - L : Left to Right Scanning
  - L : Leftmost derivation
  - 1 : un symbole de prévision dans le choix de la règle à utiliser
  - Grammaire dont la table d'analyse n'a pas d'entrée définie de façon multiple
- Propriétés :
  - Aucune grammaire ambiguë ou récursive à gauche ne peut être LL(1)
  - Une grammaire est LL(1) ssi pour toute $A \rightarrow \alpha \mid \beta$, $\alpha \neq \beta$
    - Pour aucun terminal a, $\alpha \Rightarrow^* a\gamma$ et $\beta \Rightarrow^* a\delta$
    - Au plus une des chaînes α, β peut se dériver en ε
    - Si $\beta \Rightarrow^* \varepsilon$ alors α ne se dérive pas en une chaîne commençant par un terminal de Suivant(A)

Analyse ascendante...

Analyse décalage-réduction

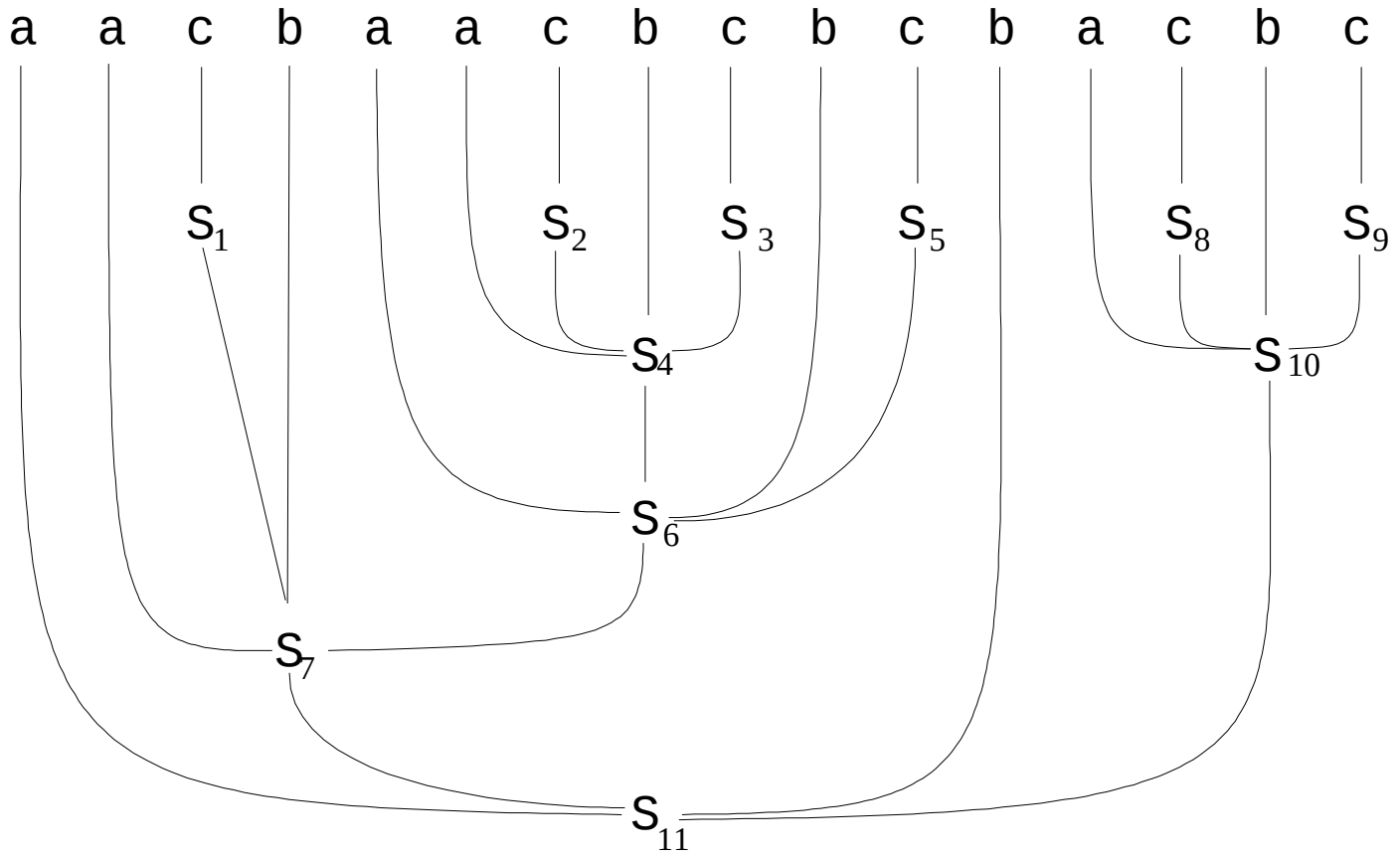
- But :
  - Construire un arbre d'analyse pour une chaîne source en commençant par les feuilles :
    - Peut être considérée comme la réduction d'un chaîne w vers l'axiome de la grammaire
    - A chaque étape de réduction, une sous-chaîne particulière correspondant à la partie droite d'une production est remplacée par le symbole de la partie gauche

Exemple...

S → aSbS | c avec le mot u=aacbaacbcbcbacbc

aacbaacbcbcbacbc	on ne peut rien **réduire**, donc on **décale**
a**a**cbaacbcbcbacbc	on ne peut rien réduire, donc on décale
aa**c**baacbcbcbacbc	on peut réduire S → c
aa**S**baacbcbcbacbc	on ne peut rien réduire, donc on décale
aaS**b**aacbcbcbacbc	on ne peut rien réduire, donc on décale
aaSb**a**acbcbcbacbc	on ne peut rien réduire, donc on décale
aaSba**a**cbcbcbacbc	on ne peut rien réduire, donc on décale
aaSbaa**c**bcbcbacbc	on peut réduire S → c
aaSbaa**S**bcbcbacbc	on ne peut rien réduire, donc on décale
aaSbaaS**b**cbcbacbc	on ne peut rien réduire, donc on décale
aaSbaaSb**c**bcbacbc	on peut réduire S → c
aaSbaaSb**S**bcbacbc	on peut réduire S → aSbS
aaSba**S**bcbacbc	on ne peut rien réduire, donc on décale
aaSbaS**b**cbacbc	on ne peut rien réduire, donc on décale
aaSbaSb**c**bacbc	on peut réduire S → c
aaSbaSb**S**bacbc	on peut réduire S → aSbS
aaSb**S**bacbc	on peut réduire S → aSbS
a**S**bacbc	on ne peut rien réduire, donc on décale
aS**b**acbc	on ne peut rien réduire, donc on décale
aSb**a**cbc	on ne peut rien réduire, donc on décale
aSba**c**bc	on peut réduire S → c
aSba**S**bc	on ne peut rien réduire, donc on décale
aSbaS**b**c	on ne peut rien réduire, donc on décale
aSbaSb**c**	on ne peut rien réduire, donc on décale
aSbaSb**c**	on peut réduire S → c
aSbaSb**S**	on peut réduire S → aSbS
aSb**S**	on peut réduire S → aSbS
S	TERMINE

Exemple...

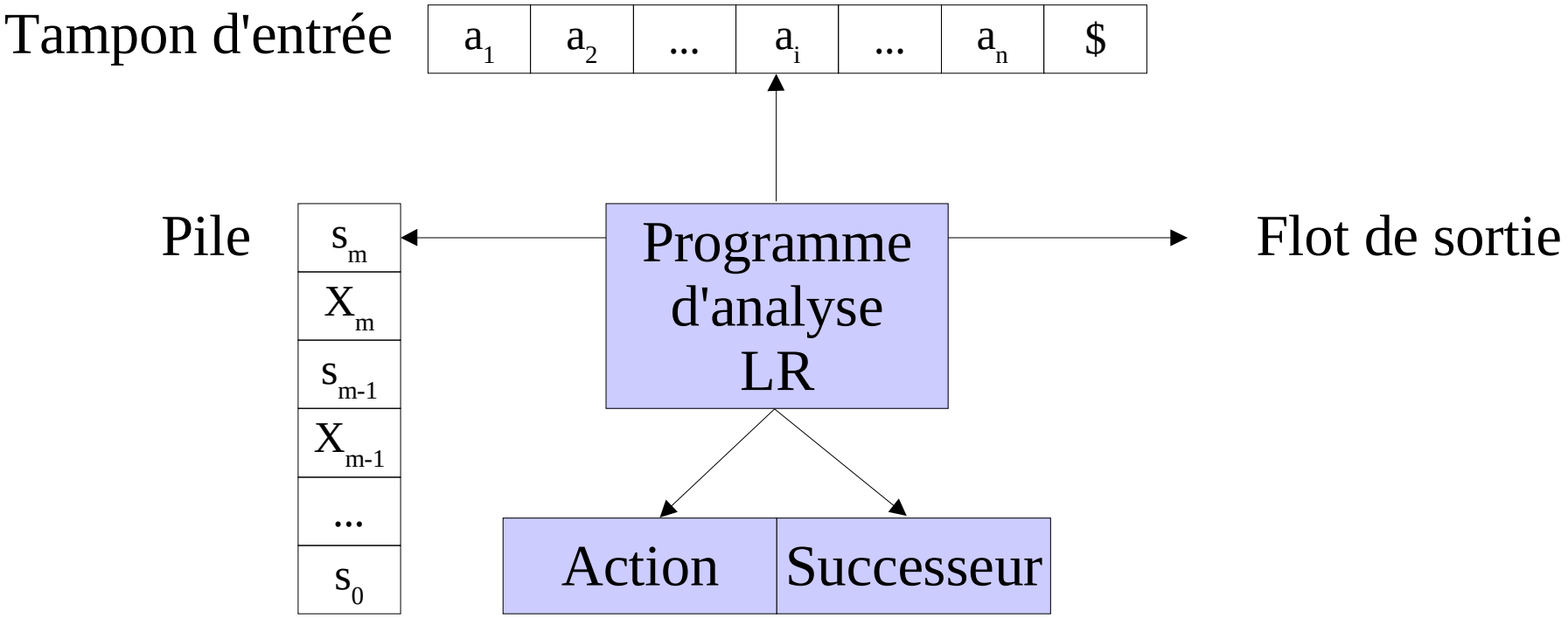


Analyseurs LR(k)...

- Définition
  - Analyseur syntaxique ascendant
  - L : Left To Right Scanning
  - R : Rightmost derivation
  - k : k symboles de prévision
- Caractéristiques :
  - Analyse toute grammaire non contextuelle
  - Implémentation efficace
  - Détection des erreurs de syntaxe aussitôt que possible

Fonctionnement d'un analyseur LR(k)...

- Schématisé par :



Fonctionnement d'un analyseur LR(k)...

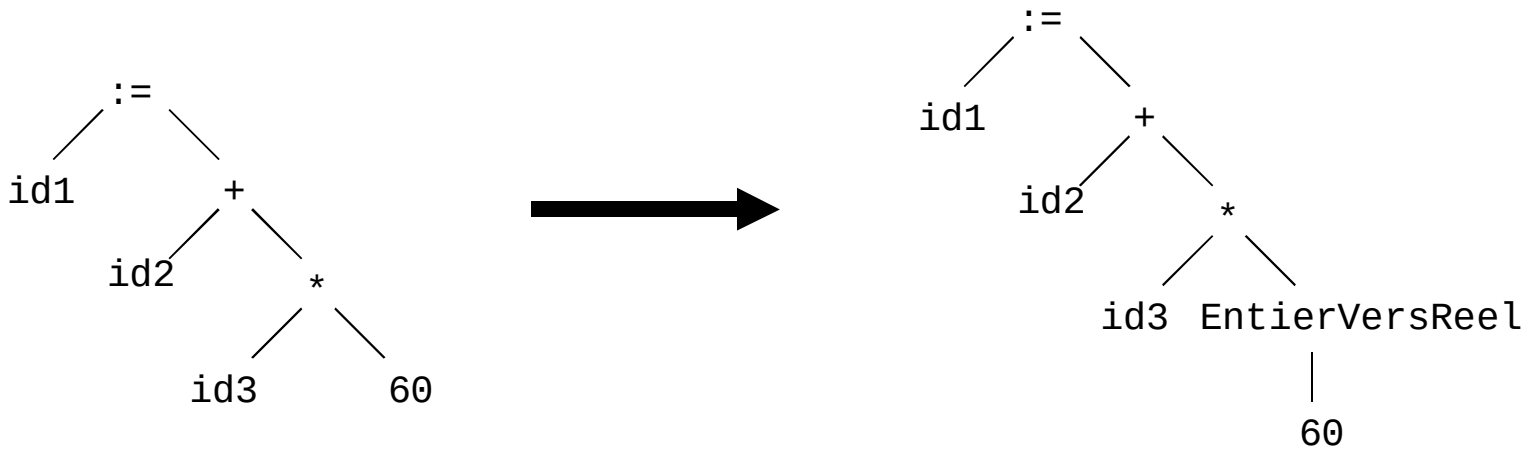
- Configuration :
  - $(s_0X_1s_1X_2..X_ms_m,a_ia_{i+1}..a_n\$)$
    - qui représente la protophrase $X_1X_2...X_ma_ia_{i+1}..a_n$
- Action[s_m,ai] =
  - décaler s : $(s_0X_1s_1X_2..X_ms_ma_is,a_{i+1}..a_n\$)$
  - réduire par $A \rightarrow \beta$: $(s_0X_1s_1X_2..X_{m-r}s_{m-r}As,a_ia_{i+1}..a_n\$)$
    - avec s=Successeur[s_{m-r},A] et r est la longueur de β
  - accepter : analyse terminée
  - erreur

Construction des tables LR...

- Il existe trois méthodes :
  - SLR (S=Simple):
    - limité par le nombre de grammaire
  - LR canonique :
    - ne peut pas être mis en oeuvre à la main
      - table d'analyse très grande
  - LALR (LA=LookAhead):
    - compromis entre les deux techniques précédentes

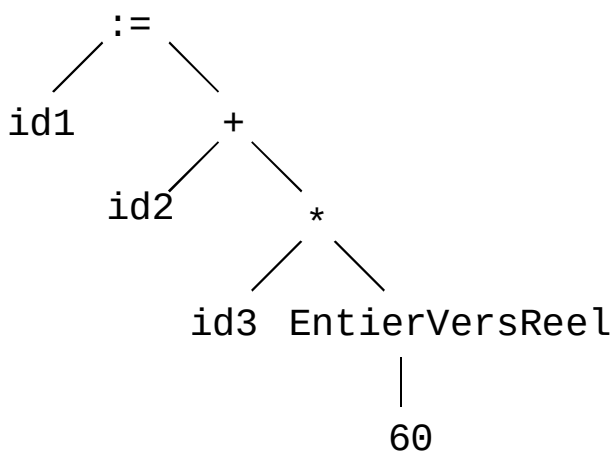
Analyse sémantique...

- Collecte d'informations destinées à la production de code
- Contrôle de type



Générateur de code intermédiaire...

- Programme pour une machine abstraite
  - Code facile à produire, à traduire en langage cible



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

Optimiseur de code...

- Optimiser le code intermédiaire pour une exécution machine plus rapide

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

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

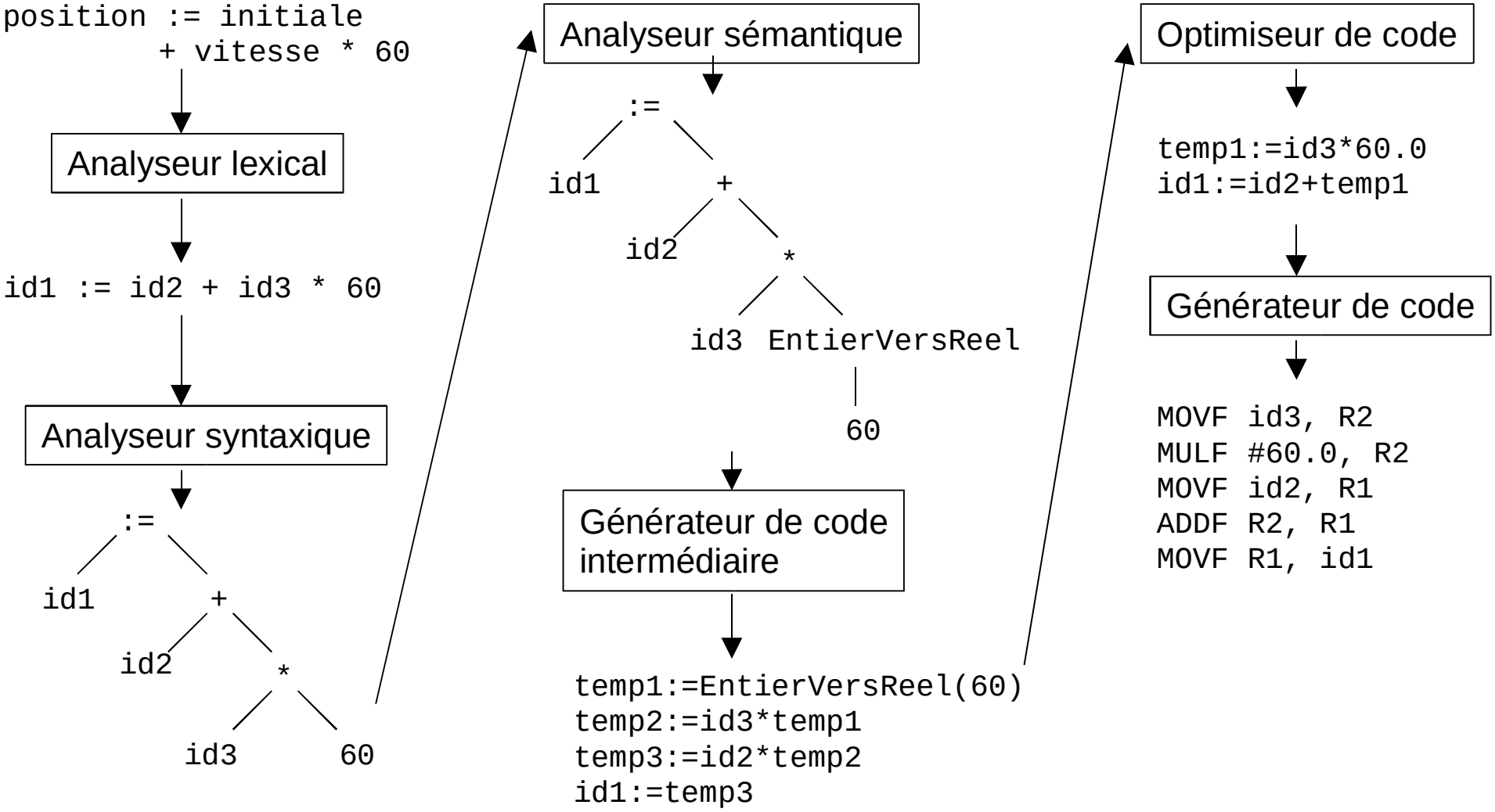
Générateur de code...

- Obtention du code machine

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

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

Récapitulatif...

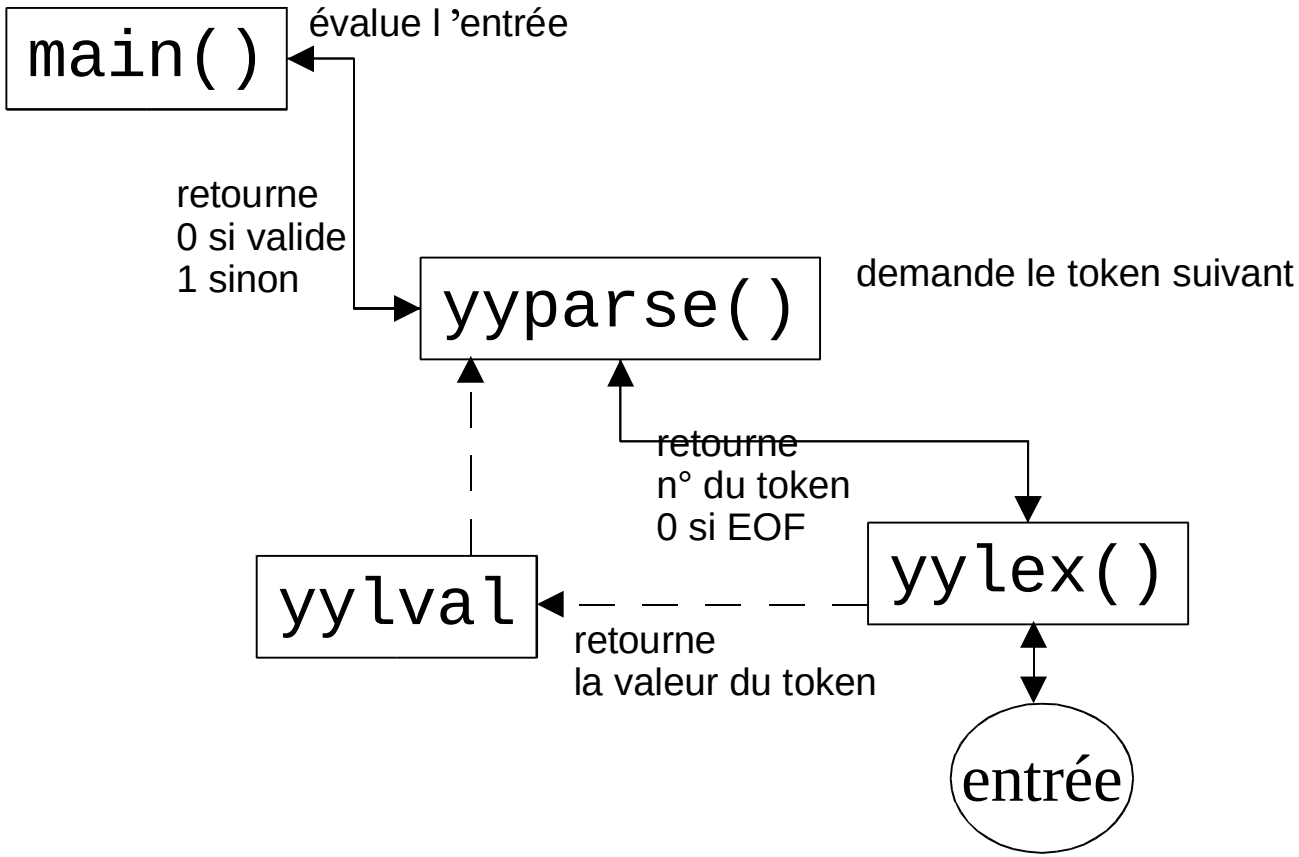


Lex et Yacc...

- Outils permettant de créer facilement des interpréteurx/compilateurs en C
- Lex :
  - Un générateur d'analyseur lexical
  - Génère entre autres une fonction C *yylex()*
- Yacc (Yet Another Compiler of Compiler)
  - Un générateur d'analyseur syntaxique
  - Génère entre autres une fonction C *yyparse()*

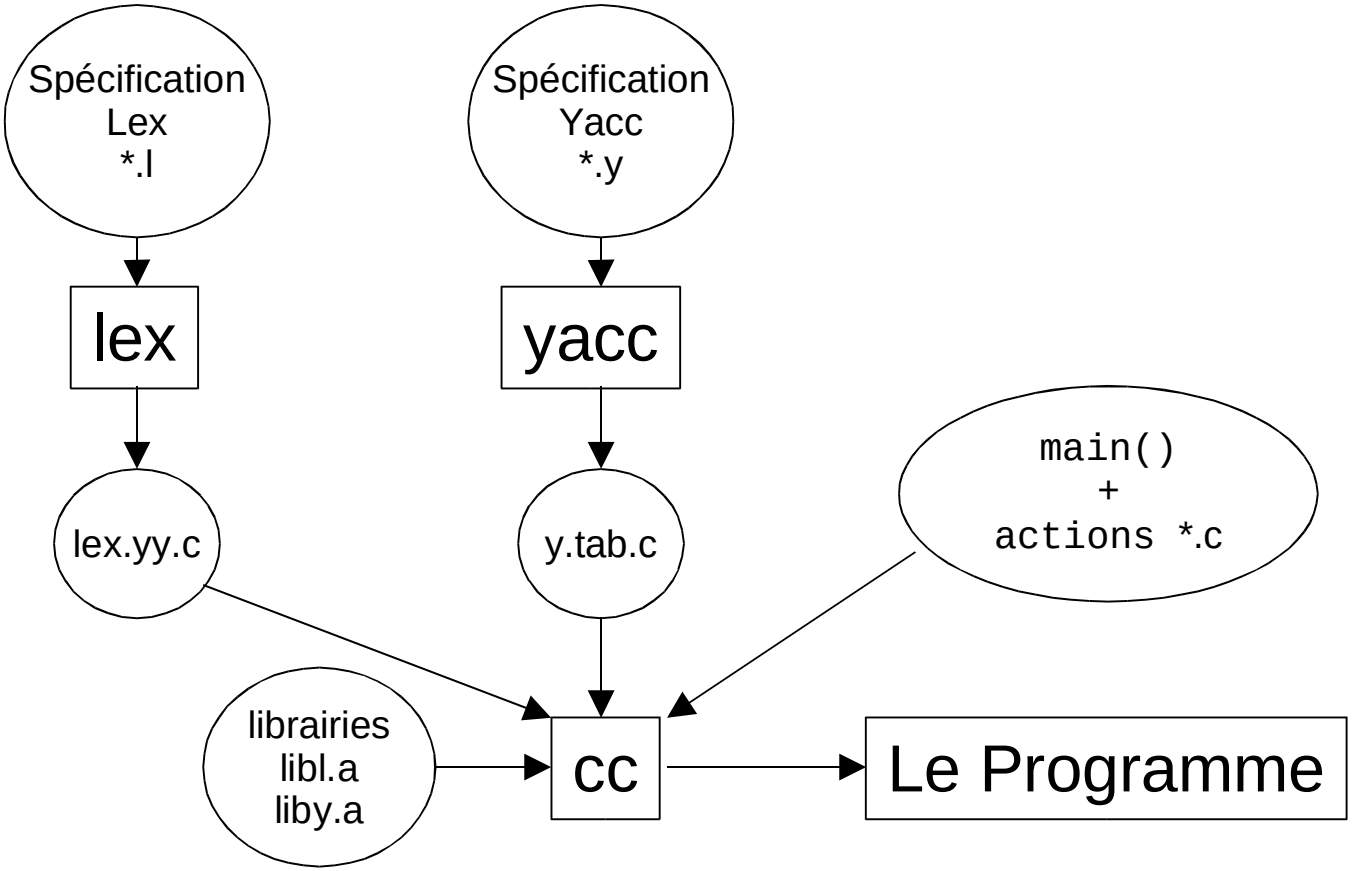
Lex et Yacc...

- Collaboration de Lex et Yacc



Lex et Yacc...

- Utilisation de Lex et Yacc



Lex : structure du fichier source...

```
%{ déclarations C
}%
définitions :
<identificateurs>    <expression régulière>
...

%%

règles : <expression régulières> <commande en C>

%%
<programme principal et sous-programmes>
```

Lex : variables et fonctions prédéfinies...

- Le programme C produit par Lex contient un certain nombre de variables et de fonctions prédéfinies que l'on peut utiliser :
  - `int yylex()` invoque le lexer et retourne le prochain lexème
  - `char* yytext` mot courant
  - `yyleng` longueur du lexème courant
  - `yylval` valeur associé au lexème
  - `FILE*` yyout fichier de sortie
  - `FILE*` yyin fichier d'entré

Lex : exemple...

```
%{
/*
** Analyseur lexical pour la construction d'une calculatrice à mémoire
** ENSICA, SSI 1999
** Hugues Cassé
** http://www.irit.fr/ACTIVITES/EQ_HECTOR/casse/alcatel/
*/

#include <string.h>
#include <stdlib.h>
#include "y.tab.h"
char *stocker_chaine(char *);
%}

ENT             [0-9]+
EXP             [+-][eE]{ENT}
VIRG            \.{ENT}
NBRE            {ENT}|{ENT}{VIRG}|{ENT}{EXP}|{ENT}{VIRG}{EXP}
IDENT           [_a-zA-Z][_0-9a-zA-Z]*
```

Lex : exemple...

```
%%
[ \t\n]         ;
"let"           return LET;
"exit"          return EXIT;
{IDENT}         {yylval.ch = stocker_chaine(yytext); return ID;}
{NBRE}          {yylval.val = atof(yytext); return NB;}
[+\-*/^]        return *yytext;
[()=;]          return *yytext;


%%

char *stocker_chaine(char *ch)
{
                char *nch = malloc(strlen(ch)+1);
                strcpy(nch, ch);
                return ch;
}
```

Yacc : Structure du fichier source...

```
%{ déclarations C
}%
définitions
%%

//productions de la grammaire
production  :     liste de symboles {action sémantique}
            |     liste de symboles {action sémantique}
            ...
            ;

%%
<programme principal et sous-programmes>
```

Yacc : exemple...

```
%{
#include <stdio.h>
#include <math.h>

double trouver_var(char *id);
void creer_var(char *id, double val);
%}

%union {double val;char *ch;}   // Définit le type de yylval (nommé YYSTYPE)
%token <val> NB                 // Indique le champ qui doit être utilisé dans
%token <ch> ID                  // l'interprétation des règles de la grammaire
%token LET EXIT
%left '+' '-'                   // Définit l'ordre des priorités ainsi que
%left '*' '/'                  // l'associativité
%right '^'
%type <val> calcul              // Indique le champ qui doit être utilisé dans
                                // un non terminal de la grammaire
```

Yacc : exemple...

```
%%

lignes: ligne
        |lignes ';' ligne
        ;

ligne:  calcul                  {printf("%lf\n", $1);}
        |LET ID '=' calcul      {creer_var($2, $4);}
        |EXIT                   {YYACCEPT;}
        ;

calcul: ID                      {$$ = trouver_var($1);}
        |NB                     {$$ = $1;}
        |'(' calcul ')'         {$$ = $2;}
        |'+' calcul             {$$ = $2;}
        |'-' calcul             {$$ = - $2;}
        |calcul '+' calcul      {$$ = $1 + $3;}
        |calcul '-' calcul      {$$ = $1 - $3;}
        |calcul '*' calcul      {$$ = $1 * $3;}
        |calcul '/' calcul      {$$ = $1 / $3;}
        |calcul '^' calcul      {$$ = pow($1, $3);}
        ;
```

Yacc : exemple...

```
%%
yyerror(char *msg){puts(msg);}

typedef struct var {
        struct var *suiv;
        char *nom;
        double val;
} var;
var *vars = 0;

void creer_var(char *id, double val) {
        var *v;
        for(v = vars; v; v = v->suiv)
                if(! strcmp(id, v->nom))
                {       v->val = val;
                        return;
                }
        v = (var *)malloc(sizeof(var));
        v->suiv = vars;
        vars = v;
        v->nom = id;
        v->val = val;
}
```

Yacc : exemple...

```
double trouver_var(char *id){
        var *v;
        for(v = vars; v; v = v->suiv)
                if(! strcmp(id, v->nom))
                        return v->val;
        return 0;
}

int main(void){
        yyparse();
}
```

Références

- Bibliographie :

Alfred V. Aho, Ravi Sethi, Jeffrey Ullman, "Compilateurs, principes, techniques et outils", Addison Wesley 1986; ISBN 0-201-10088-6

Nino Silvero, "Réaliser un compilateur, les outils Lex et Yacc", Eyrolles, ISBN 2-212-08834-5

- Cours en ligne :

http://www.pps.jussieu.fr/~dicosmo/CourseNotes/Compilation/

http://www.univ-valenciennes.fr/limav/donsez/cours/

http://fastnet.univ-brest.fr/~gire/COURS/COMPIL_IUP/POLY.html

Références

- Lex et Yacc :

  http://epaperpress.com/y_man.html

- ANTLR :

  http://www.antlr.org/