

# Théorie des langages

## Théorie des langages et compilation

---

Jérôme Delobelle

`jerome.delobelle@u-paris.fr`

LIPADE - Université de Paris

1. Structure d'un compilateur
2. Analyse lexicale
3. Analyse syntaxique
4. Analyse sémantique
5. Conclusion

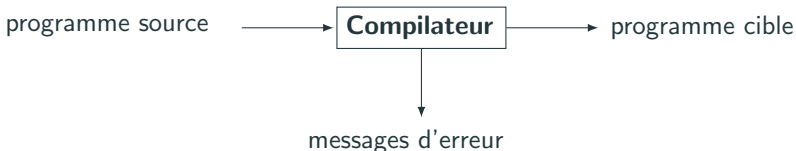
# Structure d'un compilateur

---

# Un compilateur, c'est quoi ?

Un compilateur est un **programme** qui

- prend en entrée une **donnée textuelle** source (programme, donnée xml, fichier de configuration, etc)
- la **reconnaît** (l'analyse) pour vérifier sa correction
- émet éventuellement un **message d'erreur**
- le traduit dans un **langage cible**



## Exemple

programme source

```
entier d;

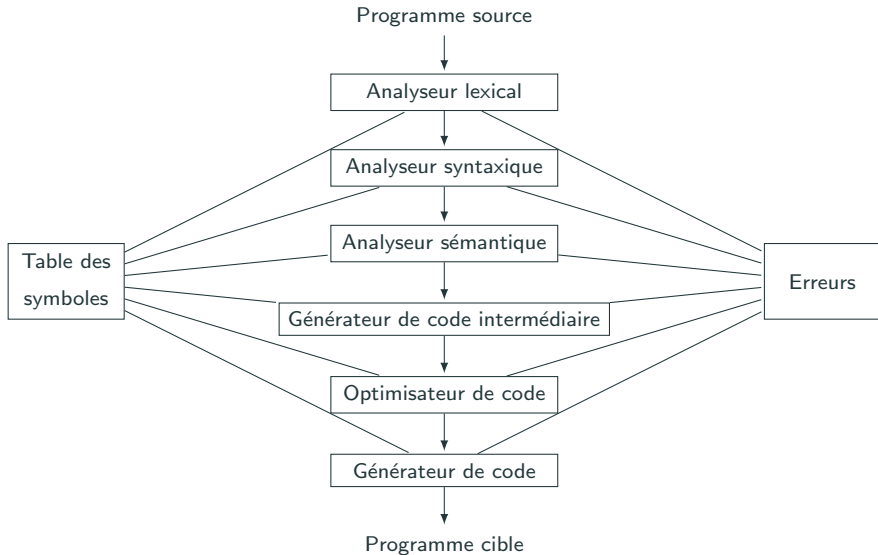
f(entier a, entier b)
entier c, entier k;
{
    k = a + b;
    retour k;
}

main()
{
    d = 7;
    ecrire(f(d, 2) + 1);
}
```

programme cible

```
f:
    push ebp
    mov  ebp, esp
    sub  esp, 8
    mov  ebx, [ebp + 12]
    push ebx
    mov  ebx, [ebp + 8]
    push ebx
    ...
main:
    push ebp
    mov  ebp, esp
    sub  esp, 8
    push 7
    pop  ebx
    mov  [d], ebx
    ...
```

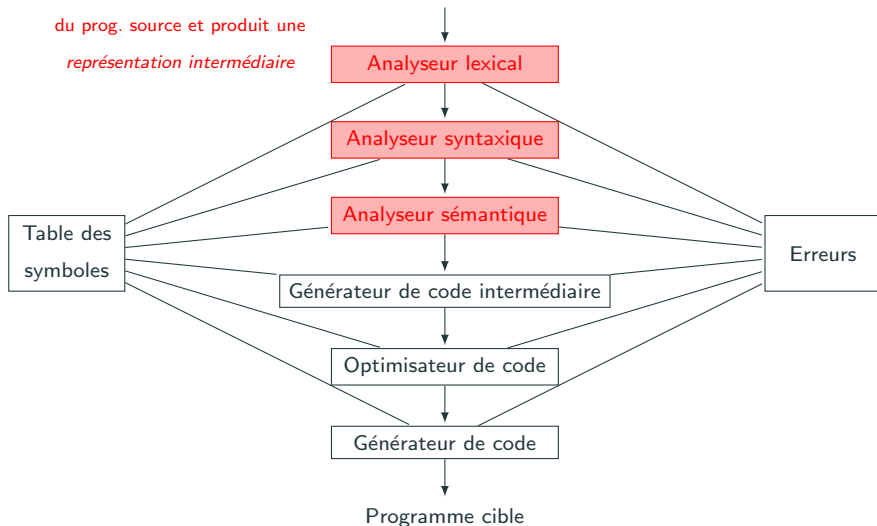
# Les différentes étapes de la compilation



# Les différentes étapes de la compilation

**Partie analyse :** sépare les  $\neq$  constituants Programme source

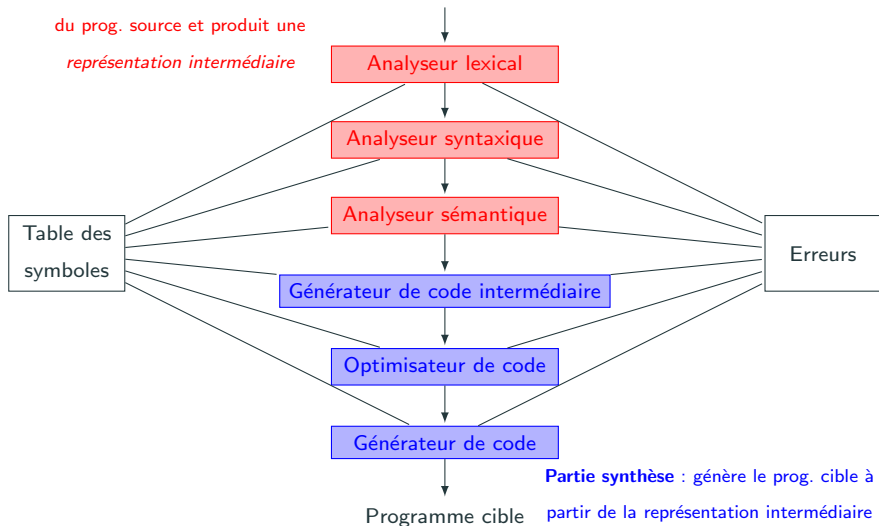
du prog. source et produit une  
*représentation intermédiaire*



# Les différentes étapes de la compilation

**Partie analyse :** sépare les  $\neq$  constituants Programme source

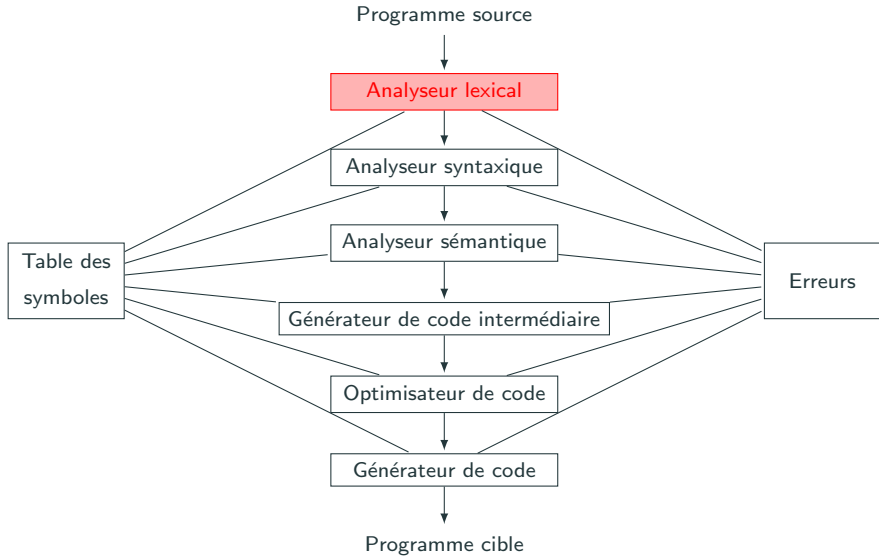
du prog. source et produit une  
*représentation intermédiaire*



**Partie synthèse :** génère le prog. cible à  
partir de la représentation intermédiaire

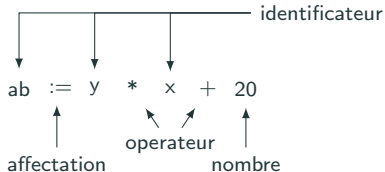


# Analyse lexicale

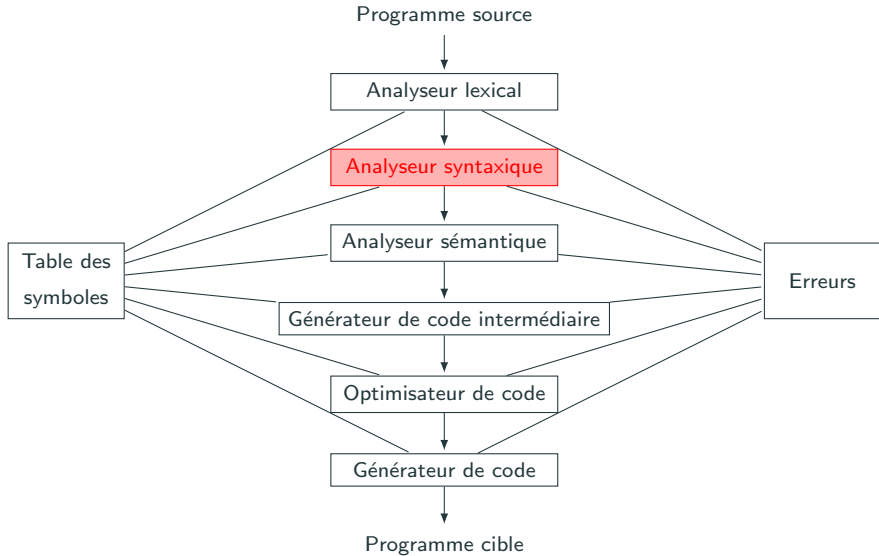


# Analyse lexicale

- Seul module au contact avec le texte source
- Son but est de reconnaître les **unités lexicales** ou **lexèmes**
  - les identificateurs et les mots clefs du langage
  - l'affectation et les opérateurs
- Utilise des expressions régulières : **automates finis**

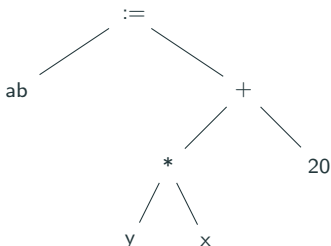


# Analyse syntaxique

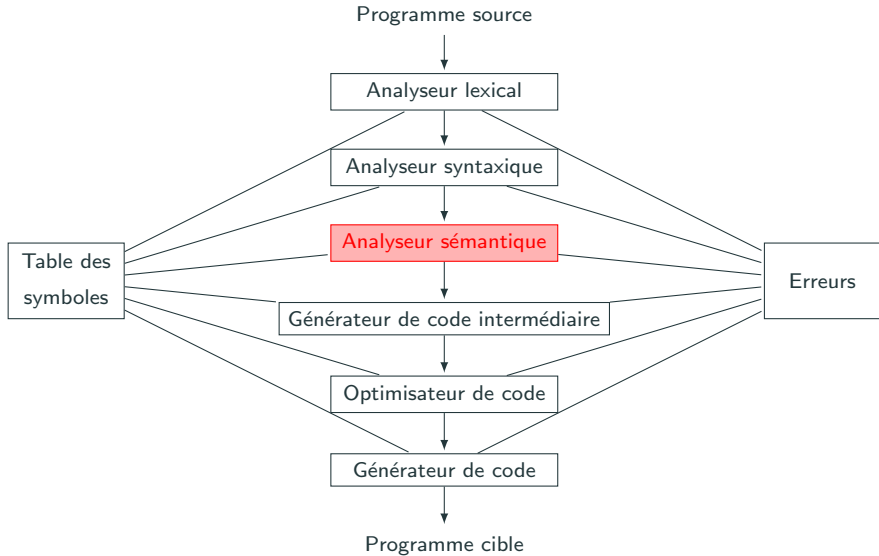


# Analyse syntaxique

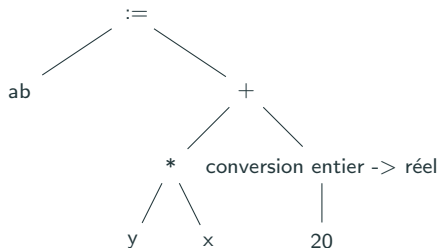
- Regroupe les unités lexicales en structures grammaticales en suivant les règles figurant dans une **grammaire**
- Résultat représenté par un arbre syntaxique
- La structure hiérarchique d'un programme est exprimée à l'aide de règles
- Grammaires hors contexte : **automates à pile**

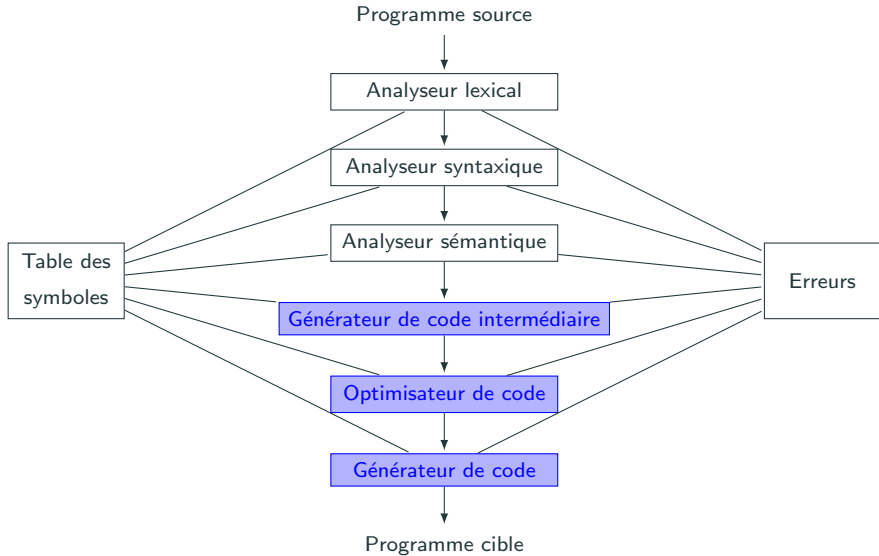


# Analyse sémantique



- Vérifie la présence d'erreurs d'ordre sémantique
  - Vérification de typage
  - Vérification des déclarations
- Par exemple, si  $x$  et  $y$  sont des réels :

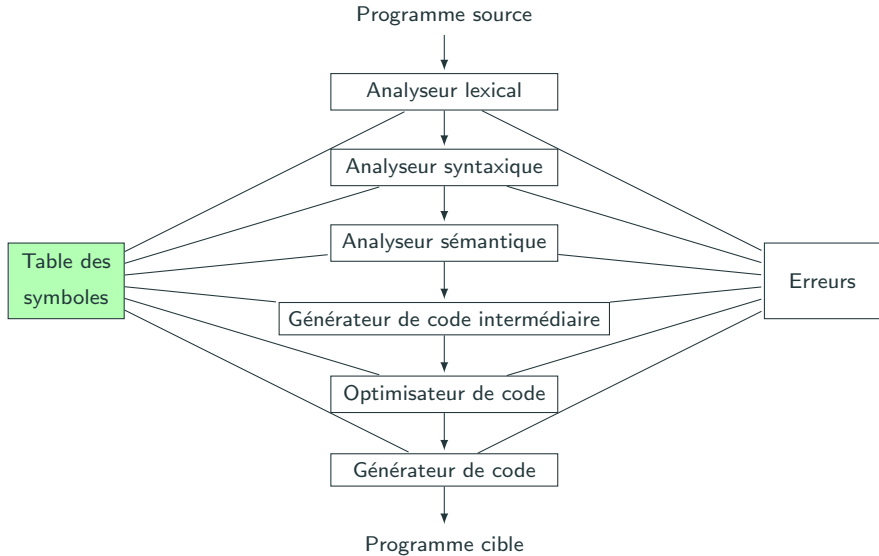




- Génération du code intermédiaire
  - Utilisation de variables temporaires
  - Choix de l'ordre pour faire un calcul
- Optimisation du code
  - Amélioration du code intermédiaire
  - Réduction du nombre de variables et d'instructions
- Génération du code
  - Choix des emplacements mémoire pour les variables
  - Assignment des variables aux registres



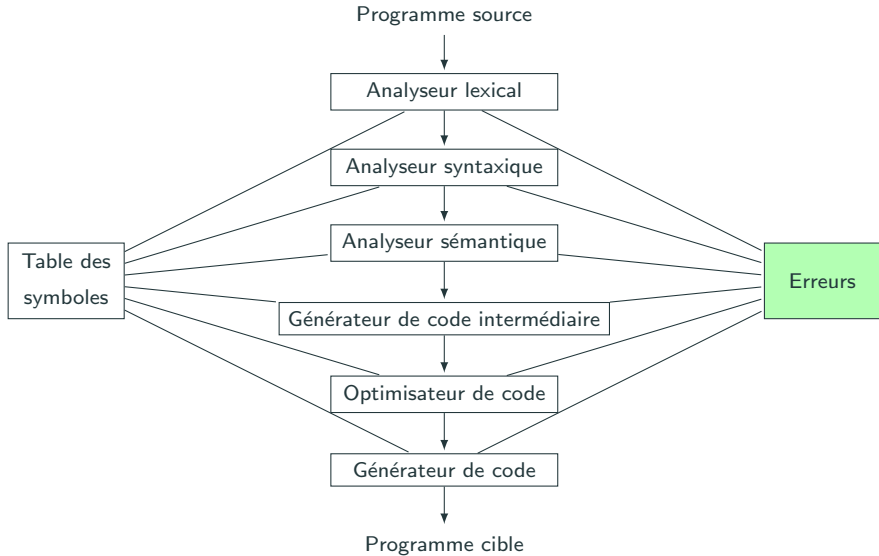
# Table des symboles



# Table des symboles

- Enregistre les identifiants et les attributs (emplacement mémoire, type, portée)
- Chaque identifiant (variable) a une entrée dans la table des symboles
- L'analyseur lexical crée une entrée dans la table des symboles à chaque fois qu'il rencontre un nouvel identificateur
  - Par contre, les attributs seront calculés plus tard
- L'analyseur sémantique se sert de la table des symboles pour vérifier la concordance des types

# Détection des erreurs



- Erreur lexicale : le flot de caractères n'est pas reconnu
- Erreur syntaxique : construction non reconnue par le langage
- Erreur sémantique : problème de typage,...

# Analyse lexicale

---

L'analyse lexicale découpe le code source en "mots" appelés **lexèmes** (ou unités lexicales) pour faciliter le travail de l'analyse syntaxique.

- Chaque lexème est un mot d'un langage représenté par une **expression régulière**
- Utilise des **automates finis** pour les reconnaître
  - Générés à partir des expressions régulières
- Ignore le texte inutile à la machine (commentaires, espaces)
- Pour chaque lexème, l'analyseur lexical émet un couple

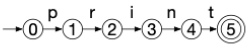

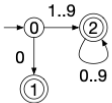
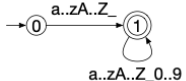
*⟨ type du lexème, valeur du lexème ⟩*

- Exemple :

*⟨ NOMBRE, 404 ⟩*

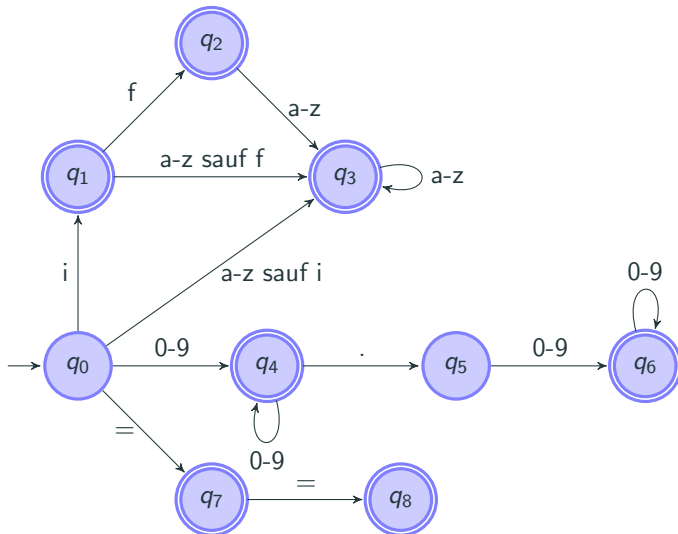
- Les types de lexèmes seront les **symboles terminaux** de la grammaire du langage lors de l'analyse syntaxique

# Reconnaissance des lexèmes

Token	Expression rationnelle	Automate
mot clé print	print	
symbole « + »	+	
nombre	$0 \mid [1-9][0-9]^*$	
identificateur	$[a-zA-Z_][a-zA-Z_0-9]^*$	

# Fragment d'automate lexical

L'analyseur lexical est donc un (gros) **automate fini déterministe** qui reconnaît l'union des langages des lexèmes





- Il existe des langages où une forme lexicale peut correspondre à différents lexèmes selon le contexte
  - **selon le contexte gauche** : par exemple, **if** reconnu comme mot-clé uniquement s'il est au début d'une instruction
  - **selon le contexte droit** : par exemple **if** reconnu comme mot-clé uniquement s'il est suivi d'une parenthèse
  - Combinaison des deux

# Problèmes de la reconnaissance contextuelle

- Dans des langages mal définis
- Pour les langages de programmation, dans des langages anciens (Fortran, partiellement C, ...)
- Ils ne peuvent en général pas être résolus par le seul analyseur lexical : les contextes sont le plus souvent des langages non-contextuels reconnus par l'analyseur syntaxique
  - Donc forte interaction entre les deux analyseurs
    - Entraîne des problèmes de lisibilité, de fiabilité et des difficultés à la maintenance et aux extensions
- C'est pourquoi la plupart des langages modernes de programmation évitent ces questions de reconnaissance contextuelle

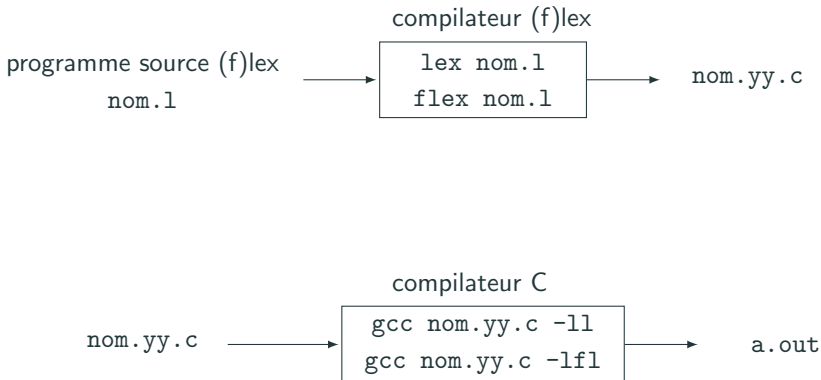
# Exemple de problème de la reconnaissance contextuelle

- en Fortran, les espaces sont non significatifs
  - $DO12i = 1.25$  : identificateur  $DO12i$ , opérateur d'affectation  $=$ , réel 1.25
  - $DO12i = 1,25$  : mot-clé  $DO$ , étiquette 12, variable  $i$  : boucle qui se termine à l'énoncé étiqueté 12 pour  $i$  allant de 1 à 25
  - d'après la légende, on aurait perdu une fusée du projet Mercury à cause d'un '.' à la place d'une ',' (à moins que ce ne soit l'inverse...)
- Dans la plupart des langages de programmation actuels, les espaces sont significatifs, et les mots-clés sont réservés

- Langage C : *lex* ou *flex*
- Langage Ada : *Alex*
- Langage Java : *Jflex*
- Langage OCaml : *OCamllex*

# Construire un analyseur lexical avec lex

lex est un utilitaire Unix produisant des analyseurs lexicaux reconnaissant des expressions régulières décrites dans un fichier.



*déclarations*

*%%*

*règles de traduction*

*%%*

*procédures auxiliaires*

# Déclarations

La section des **déclarations** comprend des

- déclarations de variables et de constantes littérales,
- définitions régulières qui permet d'associer un nom à une expression régulière

```
%{
```

```
/* définitions des constantes littérales */
```

```
PPQ, PPE, EGA, DIF, PGQ, PGE, SI, ALORS, SINON, ID, NB, OPREL
```

```
%}
```

```
/* définitions régulières */
```

```
delim      [ \n\t]
```

```
bl         {delim}+
```

```
lettre     [A-Za-z]
```

```
chiffre    [0-9]
```

```
id         {lettre}+({lettre}|{chiffre})*
```

```
nombre     {chiffre}+(\.{chiffre}+)?(E[+\-]?{chiffre}+)?
```

```
%%
```

Les **règles de traduction** sont des instructions de la forme :

$exp_1$	$action_1$
$exp_2$	$action_2$
...	
$exp_n$	$action_n$

- chaque  $exp_i$  est une expression régulière
- chaque  $action_i$  est une suite d'instruction en C qui décrit quelle action l'analyseur lexical devrait réaliser quand un lexème concorde avec le modèle  $exp_i$



## Exemple – règles de traduction

```
{bl}          { /* pas d'action et pas de retour */ }
si            { return(SI); }
alors         { return(ALORS); }
sinon         { return(SINON); }
{id}          {yyval = RangerId(); return(ID); }
{nombre}      {yyval = RangerNb(); return(NB); }
"<"          {yyval = PPQ; return(OPREL); }
"<="         {yyval = PPE; return(OPREL); }
"="           {yyval = EGA; return(OPREL); }
"<>"         {yyval = DIF; return(OPREL); }
">"          {yyval = PGQ; return(OPREL); }
">="         {yyval = PGE; return(OPREL); }
```

%%

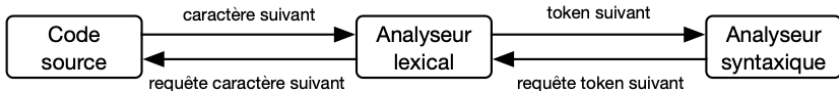
Les **procédures auxiliaires** sont les fonctions **facultatives** qui pourraient être utiles dans les actions.

```
RangerId() {  
    /* Procédure pour ranger dans la table des symboles le lexème  
    dont le premier caractère est pointé par yytext et dont la  
    longueur est yyleng et retourner un pointeur sur son entrée */  
}
```

```
RangerNb() {  
    /* procédure similaire pour ranger un lexème qui est un  
    nombre */  
}
```

# Analyseur lexical et analyseur syntaxique

- Analyseur lexical activé par l'analyseur syntaxique :
  - Lit le texte d'entrée, caractère par caractère
  - Cherche le plus long préfixe du texte d'entrée qui corresponde à l'une des expressions régulières  $exp_i$
  - Exécute alors l'action  $action_i$
- Action : rend le contrôle à l'analyseur syntaxique
- Sinon, l'analyseur lexical continue à chercher d'autres lexèmes jusqu'à ce qu'une action rende le contrôle à l'analyseur syntaxique
- Seule information retournée à l'analyseur syntaxique : **l'unité lexicale**
- Pour passer une valeur d'attribut donnant des informations sur le lexème, on peut l'affecter à une variable globale appelée `yyval`.



# Analyse syntaxique

---

# Qu'est ce que l'analyse syntaxique ?

- L'analyse syntaxique a pour but :
  - de vérifier que le texte d'entrée est conforme à la grammaire du langage source
  - d'indiquer les erreurs de syntaxe
  - de construire une représentation intermédiaire pour les autres modules du compilateur

→ **arbre de syntaxe abstraite**, utilisé par la suite de la compilation
- Ne s'occupe pas des aspects contextuels de la syntaxe
  - par exemple détecter l'erreur dans `int i; ...; i = "abcd";`
  - c'est le rôle de l'*analyse sémantique*, qui travaille sur la représentation intermédiaire
- Représentation unique du texte source → l'analyse doit être déterministe, si possible linéaire (éviter notamment les retours en arrière)

- **Analyseur syntaxique descendant**, ou **analyseur prédictif**
  - Construit l'arbre de dérivation à **partir de sa racine** et en effectuant des dérivations en considérant **la tête des règles** de production et en faisant des dérivations **les plus à gauche**
  - Famille des analyseurs **LL** (**left scanning, leftmost derivation**)
  - JavaCC, ANTLR, LLGen, ...
- **Analyseur syntaxique ascendant** ou par **décalage-réduction**
  - Construit l'arbre de dérivation à **partir de ses feuilles** et en effectuant des dérivations en considérant **la partie droite des règles** de production et en faisant des dérivations **les plus à gauche**
  - Famille des analyseurs **LR** (**left scanning, rightmost derivation**)
  - YACC/Bison

## Exemple

Soit  $G = \langle V, \Sigma, P, S \rangle$  avec

- $\Sigma = \{a, b, c, d\}$
- $V \setminus \Sigma = \{S, T\}$
- 6 règles de production :
  - $S \rightarrow aSbT|cT|d$
  - $T \rightarrow aT|bS|c$

## Exemple

Soit  $G = \langle V, \Sigma, P, S \rangle$  avec

- $\Sigma = \{a, b, c, d\}$
- $V \setminus \Sigma = \{S, T\}$
- 6 règles de production :
  - $S \rightarrow aSbT|cT|d$
  - $T \rightarrow aT|bS|c$
- Soit  $w = accbbadbc$



## Exemple

Soit  $G = \langle V, \Sigma, P, S \rangle$  avec

- $\Sigma = \{a, b, c, d\}$
- $V \setminus \Sigma = \{S, T\}$
- 6 règles de production :
  - $S \rightarrow aSbT|cT|d$
  - $T \rightarrow aT|bS|c$
- Soit  $w = accbbadbcb$

**Analyse LL :**

$S$

## Exemple

Soit  $G = \langle V, \Sigma, P, S \rangle$  avec

- $\Sigma = \{a, b, c, d\}$
- $V \setminus \Sigma = \{S, T\}$
- 6 règles de production :
  - $S \rightarrow aSbT|cT|d$
  - $T \rightarrow aT|bS|c$
- Soit  $w = accbbadbcb$

**Analyse LL :**

$$S \rightarrow aSbT$$

## Exemple

Soit  $G = \langle V, \Sigma, P, S \rangle$  avec

- $\Sigma = \{a, b, c, d\}$
- $V \setminus \Sigma = \{S, T\}$
- 6 règles de production :
  - $S \rightarrow aSbT|cT|d$
  - $T \rightarrow aT|bS|c$
- Soit  $w = accbbadbcb$

**Analyse LL :**

$$S \rightarrow aSbT \rightarrow ac**T**bT$$

## Exemple

Soit  $G = \langle V, \Sigma, P, S \rangle$  avec

- $\Sigma = \{a, b, c, d\}$
- $V \setminus \Sigma = \{S, T\}$
- 6 règles de production :
  - $S \rightarrow aSbT|cT|d$
  - $T \rightarrow aT|bS|c$
- Soit  $w = accbbadbc$

**Analyse LL :**

$$S \rightarrow aSbT \rightarrow acTbT \rightarrow accb\textcolor{red}{T}$$

## Exemple

Soit  $G = \langle V, \Sigma, P, S \rangle$  avec

- $\Sigma = \{a, b, c, d\}$
- $V \setminus \Sigma = \{S, T\}$
- 6 règles de production :
  - $S \rightarrow aSbT|cT|d$
  - $T \rightarrow aT|bS|c$
- Soit  $w = accbbadbc$

**Analyse LL :**

$$S \rightarrow aSbT \rightarrow acTbT \rightarrow accbT \rightarrow accbbS$$

## Exemple

Soit  $G = \langle V, \Sigma, P, S \rangle$  avec

- $\Sigma = \{a, b, c, d\}$
- $V \setminus \Sigma = \{S, T\}$
- 6 règles de production :
  - $S \rightarrow aSbT|cT|d$
  - $T \rightarrow aT|bS|c$
- Soit  $w = accbbadbc$

**Analyse LL :**

$S \rightarrow aSbT \rightarrow acTbT \rightarrow accbT \rightarrow accbbS \rightarrow accbbaSbT$

## Exemple

Soit  $G = \langle V, \Sigma, P, S \rangle$  avec

- $\Sigma = \{a, b, c, d\}$
- $V \setminus \Sigma = \{S, T\}$
- 6 règles de production :
  - $S \rightarrow aSbT|cT|d$
  - $T \rightarrow aT|bS|c$
- Soit  $w = accbbadbc$

**Analyse LL :**

$S \rightarrow aSbT \rightarrow acTbT \rightarrow accbT \rightarrow accbbS \rightarrow accbbaSbT$   
 $\rightarrow accbbadbT$

## Exemple

Soit  $G = \langle V, \Sigma, P, S \rangle$  avec

- $\Sigma = \{a, b, c, d\}$
- $V \setminus \Sigma = \{S, T\}$
- 6 règles de production :
  - $S \rightarrow aSbT|cT|d$
  - $T \rightarrow aT|bS|c$
- Soit  $w = accbbadbc$

**Analyse LL :**

$$\begin{aligned} S &\rightarrow aSbT \rightarrow acTbT \rightarrow accbT \rightarrow accbbS \rightarrow accbbaSbT \\ &\rightarrow accbbadbT \rightarrow accbbadbc \end{aligned}$$



## Exemple

Soit  $G = \langle V, \Sigma, P, S \rangle$  avec

- $\Sigma = \{a, b, c, d\}$
- $V \setminus \Sigma = \{S, T\}$
- 6 règles de production :
  - $S \rightarrow aSbT|cT|d$
  - $T \rightarrow aT|bS|c$
- Soit  $w = accbbadbcb$

**Analyse LL :**

$$\begin{aligned} S &\rightarrow aSbT \rightarrow acTbT \rightarrow accbT \rightarrow accbbS \rightarrow accbbaSbT \\ &\rightarrow accbbadbT \rightarrow accbbadbcb \end{aligned}$$

**Analyse LR :**

*accbbadbcb*

## Exemple

Soit  $G = \langle V, \Sigma, P, S \rangle$  avec

- $\Sigma = \{a, b, c, d\}$
- $V \setminus \Sigma = \{S, T\}$
- 6 règles de production :
  - $S \rightarrow aSbT|cT|d$
  - $T \rightarrow aT|bS|c$
- Soit  $w = accbbadbcb$

**Analyse LL :**

$$\begin{aligned} S &\rightarrow aSbT \rightarrow acTbT \rightarrow accbT \rightarrow accbbS \rightarrow accbbaSbT \\ &\rightarrow accbbadbT \rightarrow accbbadbcb \end{aligned}$$

**Analyse LR :**

*a***c**cbbadbc

## Exemple

Soit  $G = \langle V, \Sigma, P, S \rangle$  avec

- $\Sigma = \{a, b, c, d\}$
- $V \setminus \Sigma = \{S, T\}$
- 6 règles de production :
  - $S \rightarrow aSbT \mid cT \mid d$
  - $T \rightarrow aT \mid bS \mid c$
- Soit  $w = accbbadbcb$

**Analyse LL :**

$$\begin{aligned} S &\rightarrow aSbT \rightarrow acTbT \rightarrow accbT \rightarrow accbbS \rightarrow accbbaSbT \\ &\rightarrow accbbadbT \rightarrow accbbadbcb \end{aligned}$$

**Analyse LR :**

*a***c**cbbadbcb

$\Rightarrow$  mène à un échec

## Exemple

Soit  $G = \langle V, \Sigma, P, S \rangle$  avec

- $\Sigma = \{a, b, c, d\}$
- $V \setminus \Sigma = \{S, T\}$
- 6 règles de production :
  - $S \rightarrow aSbT|cT|d$
  - $T \rightarrow aT|bS|c$
- Soit  $w = accbbadbcb$

**Analyse LL :**

$$\begin{aligned} S &\rightarrow aSbT \rightarrow acTbT \rightarrow accbT \rightarrow accbbS \rightarrow accbbaSbT \\ &\rightarrow accbbadbT \rightarrow accbbadbcb \end{aligned}$$

**Analyse LR :**

*accbbadbcb*

## Exemple

Soit  $G = \langle V, \Sigma, P, S \rangle$  avec

- $\Sigma = \{a, b, c, d\}$
- $V \setminus \Sigma = \{S, T\}$
- 6 règles de production :
  - $S \rightarrow aSbT|cT|d$
  - $T \rightarrow aT|bS|c$
- Soit  $w = accbbadbcb$

**Analyse LL :**

$$\begin{aligned} S &\rightarrow aSbT \rightarrow acTbT \rightarrow accbT \rightarrow accbbS \rightarrow accbbaSbT \\ &\rightarrow accbbadbT \rightarrow accbbadbcb \end{aligned}$$

**Analyse LR :**

$ac**c**bbadbcb$

## Exemple

Soit  $G = \langle V, \Sigma, P, S \rangle$  avec

- $\Sigma = \{a, b, c, d\}$
- $V \setminus \Sigma = \{S, T\}$
- 6 règles de production :
  - $S \rightarrow aSbT|cT|d$
  - $T \rightarrow aT|bS|c$
- Soit  $w = accbbadbcb$

**Analyse LL :**

$$\begin{aligned} S &\rightarrow aSbT \rightarrow acTbT \rightarrow accbT \rightarrow accbbS \rightarrow accbbaSbT \\ &\rightarrow accbbadbT \rightarrow accbbadbcb \end{aligned}$$

**Analyse LR :**

$$accbbadbcb \leftarrow acTbbadbcb$$

## Exemple

Soit  $G = \langle V, \Sigma, P, S \rangle$  avec

- $\Sigma = \{a, b, c, d\}$
- $V \setminus \Sigma = \{S, T\}$
- 6 règles de production :
  - $S \rightarrow aSbT \mid cT \mid d$
  - $T \rightarrow aT \mid bS \mid c$
- Soit  $w = accbbadbc$

**Analyse LL :**

$$\begin{aligned} S &\rightarrow aSbT \rightarrow acTbT \rightarrow accbT \rightarrow accbbS \rightarrow accbbaSbT \\ &\rightarrow accbbadbT \rightarrow accbbadbc \end{aligned}$$

**Analyse LR :**

$$accbbadbc \leftarrow acTbbadbc \leftarrow aSbba\textcolor{red}{d}bc$$

## Exemple

Soit  $G = \langle V, \Sigma, P, S \rangle$  avec

- $\Sigma = \{a, b, c, d\}$
- $V \setminus \Sigma = \{S, T\}$
- 6 règles de production :
  - $S \rightarrow aSbT|cT|d$
  - $T \rightarrow aT|bS|c$
- Soit  $w = accbbadbcb$

**Analyse LL :**

$$\begin{aligned} S &\rightarrow aSbT \rightarrow acTbT \rightarrow accbT \rightarrow accbbS \rightarrow accbbaSbT \\ &\rightarrow accbbadbT \rightarrow accbbadbcb \end{aligned}$$

**Analyse LR :**

$$accbbadbcb \leftarrow acTbbadbcb \leftarrow aSbbadbcb \leftarrow aSbbaSbcb$$



## Exemple

Soit  $G = \langle V, \Sigma, P, S \rangle$  avec

- $\Sigma = \{a, b, c, d\}$
- $V \setminus \Sigma = \{S, T\}$
- 6 règles de production :
  - $S \rightarrow aSbT \mid cT \mid d$
  - $T \rightarrow aT \mid bS \mid c$
- Soit  $w = accbbadbcb$

**Analyse LL :**

$$\begin{aligned} S &\rightarrow aSbT \rightarrow acTbT \rightarrow accbT \rightarrow accbbS \rightarrow accbbaSbT \\ &\rightarrow accbbadbT \rightarrow accbbadbcb \end{aligned}$$

**Analyse LR :**

$$accbbadbcb \leftarrow acTbbadbcb \leftarrow aSbbadbcb \leftarrow aSbbaSbc \leftarrow aSbbaSbT$$

## Exemple

Soit  $G = \langle V, \Sigma, P, S \rangle$  avec

- $\Sigma = \{a, b, c, d\}$
- $V \setminus \Sigma = \{S, T\}$
- 6 règles de production :
  - $S \rightarrow aSbT|cT|d$
  - $T \rightarrow aT|bS|c$
- Soit  $w = accbbadbcb$

**Analyse LL :**

$$\begin{aligned} S &\rightarrow aSbT \rightarrow acTbT \rightarrow accbT \rightarrow accbbS \rightarrow accbbaSbT \\ &\rightarrow accbbadbT \rightarrow accbbadbcb \end{aligned}$$

**Analyse LR :**

$$\begin{aligned} accbbadbcb &\leftarrow acTbbadbcb \leftarrow aSbbadbcb \leftarrow aSbbaSbc \leftarrow aSbbaSbT \\ &\leftarrow aSb**bS** \end{aligned}$$

## Exemple

Soit  $G = \langle V, \Sigma, P, S \rangle$  avec

- $\Sigma = \{a, b, c, d\}$
- $V \setminus \Sigma = \{S, T\}$
- 6 règles de production :
  - $S \rightarrow aSbT|cT|d$
  - $T \rightarrow aT|bS|c$
- Soit  $w = accbbadbcb$

**Analyse LL :**

$$\begin{aligned} S &\rightarrow aSbT \rightarrow acTbT \rightarrow accbT \rightarrow accbbS \rightarrow accbbaSbT \\ &\rightarrow accbbadbT \rightarrow accbbadbcb \end{aligned}$$

**Analyse LR :**

$$\begin{aligned} accbbadbcb &\leftarrow acTbbadbcb \leftarrow aSbbadbcb \leftarrow aSbbaSbc \leftarrow aSbbaSbT \\ &\leftarrow aSbbS \leftarrow aSbT \end{aligned}$$

## Exemple

Soit  $G = \langle V, \Sigma, P, S \rangle$  avec

- $\Sigma = \{a, b, c, d\}$
- $V \setminus \Sigma = \{S, T\}$
- 6 règles de production :
  - $S \rightarrow aSbT|cT|d$
  - $T \rightarrow aT|bS|c$
- Soit  $w = accbbadbcb$

**Analyse LL :**

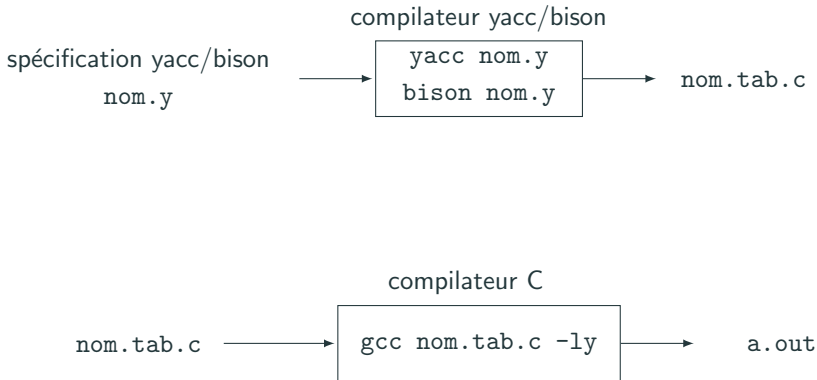
$$\begin{aligned} S &\rightarrow aSbT \rightarrow acTbT \rightarrow accbT \rightarrow accbbS \rightarrow accbbaSbT \\ &\rightarrow accbbadbT \rightarrow accbbadbcb \end{aligned}$$

**Analyse LR :**

$$\begin{aligned} accbbadbcb &\leftarrow acTbbadbcb \leftarrow aSbbadbcb \leftarrow aSbbaSbc \leftarrow aSbbaSbT \\ &\leftarrow aSbbS \leftarrow aSbT \leftarrow S \end{aligned}$$

- yacc : Yet Another Compiler Compiler (Unix)
- bison en version libre (Linux)
- Prend une grammaire hors contexte en entrée, produit l'analyseur correspondant
- Le programme produit est en langage C

# Construire un analyseur syntaxique avec yacc



*déclarations*

*%%*

*règles de production et routines sémantiques*

*%%*

*routines C et bloc principal*

La section des **déclarations** contient 2 parties optionnelles

- 1ère partie : déclarations en C
  - délimitées par %{ et %}
  - déclarations des variables temporaires utilisées par les règles de traduction
  - procédures de la deuxième et troisième section
- 2nde partie : déclarations d'unités lexicales de la grammaire
- **Symboles terminaux** :
  - **unités lexicales**, déclarées par %token  
%token NOMBRE
  - **caractères**, entre quotes : 'a', '+'
- **Symboles non terminaux** : chaînes de caractères non déclarées comme unités lexicales



- Une **règle de production** de la grammaire du type :

$\text{non-terminal} \rightarrow \text{alt}_1 \mid \text{alt}_2 \mid \dots \mid \text{alt}_n$

s'écrirait en yacc :

```
non-terminal    : alt1    {action sémantique 1}  
                  | alt2    {action sémantique 2}  
                  ...  
                  | altn    {action sémantique n}  
                  ;
```

- La partie gauche de la première règle de production est l'**axiome** de la grammaire

- Une **action sémantique** est une suite d'instructions en C
- Elles sont exécutées à chaque fois qu'il y a réduction par la production associée
- Syntaxe d'une action sémantique :
  - Symbole \$\$ référence la valeur de l'attribut associé au non terminal de la partie gauche de la règle
  - Symbole \$i représente la valeur associée au *i*ème symbole (terminal ou non) en partie droite de la règle.
  - Exemple :

```
expr    :   expr '+' term    {printf("EXPR et TERM")}  
        |   term            {printf("TERM simple")}  
        ;
```

- En langage C
- Un analyseur lexical nommé `yyllex()` doit être fourni
- D'autres procédures comme les routines de récupération d'erreur peuvent être ajoutées si nécessaire
- L'analyseur lexical `yyllex()` produit des couples formés d'une unité lexicale et de la valeur de l'attribut associé
- Si une unité lexicale est retournée, elle doit être déclarée dans la première section de la spécification `yacc`
- La valeur de l'attribut associée à une unité lexicale est communiquée à l'analyseur syntaxique par l'intermédiaire de la variable `yylval` prédéfinie dans `yacc`

# Exemple

- Grammaire des expressions arithmétiques :

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid \textit{chiffre}$$

- L'unité lexicale *chiffre* désigne un chiffre unique compris entre 0 et 9

## Exemple – déclarations et règles de production

```
%{  
    #include <ctype.h>  
%}  
%token CHIFFRE  
%%  
  
ligne      :  expr '\n'          {printf("%d\n", $1);}  
           ;  
expr       :  expr '+' term      {$$ = $1 + $3;}  
           |  term  
           ;  
term       :  term '*' facteur   {$$ = $1 * $3;}  
           |  facteur  
           ;  
facteur    :  '(' expr ')'       {$$ = $2;}  
           |  CHIFFRE  
           ;  
%%
```

## Autre exemple

```
%{  
    #include <ctype.h>  
}%  
%token INTEGER  
  
%%  
  
program    :  expr '.'          {printf("%d\n", $1);}  
           ;  
expr       :  INTEGER  
           |  expr '+' expr    {$$ = $1 + $3;}  
           |  expr '-' expr    {$$ = $1 - $3;}  
           ;  
  
%%  
  
int main(void) {  
    yyparse();  
    return 0;  
}
```

- `lex` a été conçu pour produire des analyseurs lexicaux qui peuvent être utilisés avec des analyseurs syntaxiques faits avec `yacc`
- Pour utiliser `lex`, il faut remplacer la routine `yylex()` par la clause :  
`#include "lex.yy.c"`
- Spécifier chaque action `lex` de façon à ce qu'elle retourne un terminal connu de `yacc`

- Spécification `lex` : fichier `nom.l`
- Spécification `yacc` : fichier `synt.y`
- Compilation :

```
lex nom.l
```

```
yacc synt.y
```

```
gcc -ly -ll y.tab.c
```

- On obtient un exécutable `a.out` qui permet de traduire le langage désiré



# Analyse sémantique

---

## Pourquoi ?

- Correction syntaxique insuffisante
  - Exemple :  $3 + \text{true} > 2$  est syntaxiquement correct
- ⇒ Nécessité d'une analyse sémantique pour
- Vérifier les types d'affections
  - Vérifier les signatures de fonctions + contenu (return ou non)
  - S'occuper des liaisons des variables (portée de la variable)
  - Vérifier si du code est inatteignable ou pas
  - ...

## Comment ?

- Compléter la table des symboles
- Ajouter des informations à l'arbre syntaxique abstrait

## Conclusion

---

# Conclusion

**Partie analyse :** sépare les  $\neq$  constituants Programme source

du prog. source et produit une  
*représentation intermédiaire*

