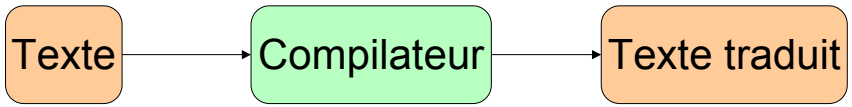
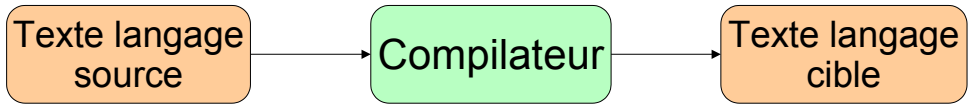
Qu'est-ce qu'un compilateur ?

Compilateur = vérificateur et traducteur

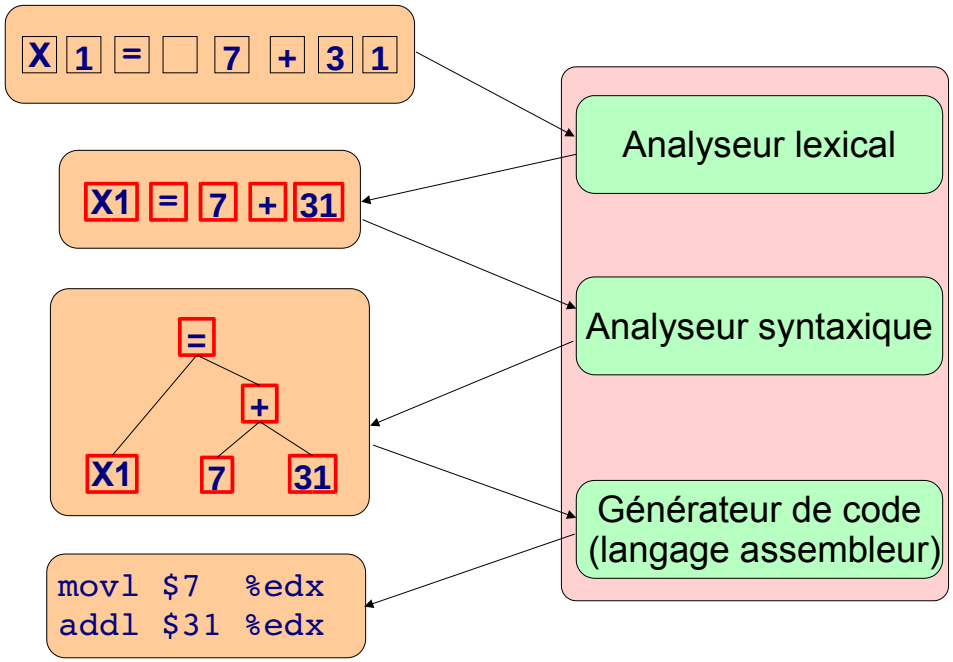


- Exemples
  - Texte en français → Texte dont l'orthographe est corrigée
  - Texte en français → Texte dont la grammaire est vérifiée
  - Programme en C → Programme en assembleur
  - Programme en assembleur → Programme en langage machine
  - Image au format JPEG → Image au format GIF

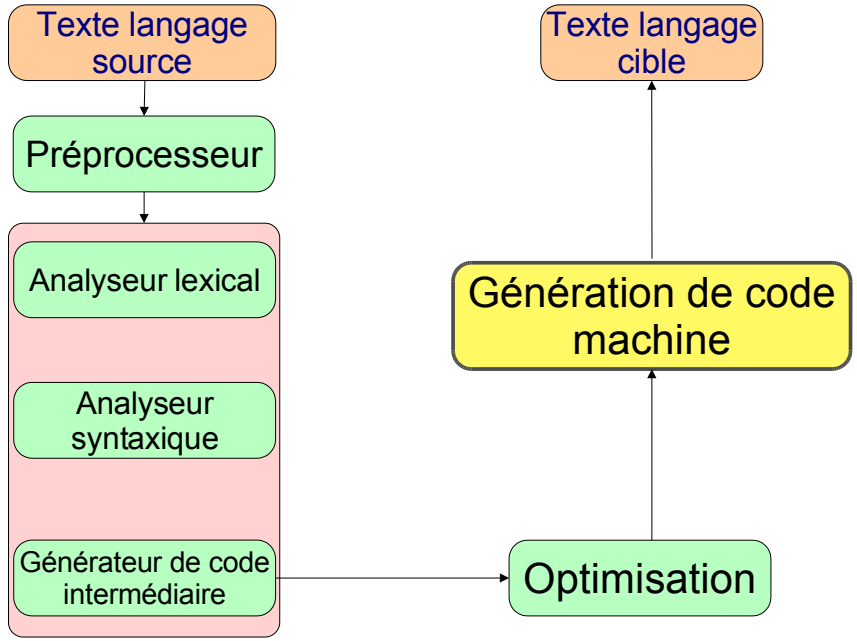


- Remarques
  - Un compilateur traduit automatiquement
  - C'est un logiciel. Exemple : gcc, p2c
  - Un compilateur travaillant sur une machine M1 peut traduire un programme en langage machine pour une autre machine M2.

Trois étapes importantes



Étapes typiques d'une compilation C

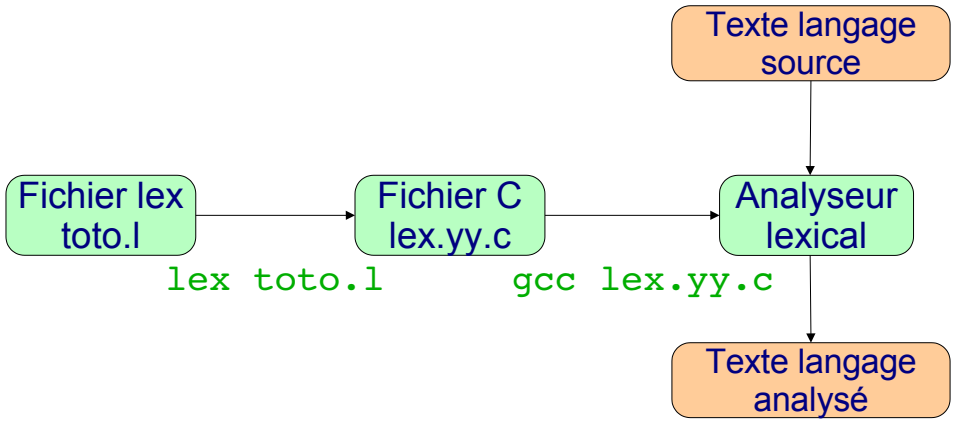


Analyse lexicale (scanners)

- Rôle : grouper les lettres pour former des mots.
- Au passage :
  - reconnaît et signale les mots mal orthographiés
  - peut supprimer les commentaires.
  - peut préprocesser le texte : expansion de macros.

Lex : un outil pour construire un analyseur lexical

- Logiciel qui construit automatiquement un analyseur lexical



Lex : fonctionnement

- On décrit un ensemble de mots par une expression
- On associe à chaque expression une action
- L'exécutable produit par lex lit le fichier d'entrée et pour chaque mot reconnu, effectue une action précisée par l'utilisateur.

Expression ⟶ **Action**

Exemple : transformer tous les `'A'` en `'a'`

```
A      {printf(''a'');}
```

Format d'un source lex

```
<partie définitions>          } Optionnel

%%

<partie règles>

%%                            }
                              } Optionnel
<partie code utilisateur (C)> }
```

Partie 1: groupe littéral

- Le texte compris entre %{ et %} est recopié par lex dans le source C généré avant **yylex()**.
- Exemple :
  - Le programmeur veut utiliser la fonction **printf** dans les actions.
  - Il a aussi besoin d'une variable **num_lignes**

```
%{
#include <stdio.h>
int num_lignes;
%}
%%
 ....partie 2....
```

Partie 1: définitions (ou *macros*)

- Définition de noms : pour définir des abréviations utilisées plus tard (dans la partie 2 (règles)).

<nom> <expression>

- Exemples :
  - `ID [A-Za-z_][A-Za-z0-9_]*`
  - Dans la partie règles, on pourra utiliser `{ID}` à la place de `[A-Za-z_][A-Za-z0-9_]*`.
  - Attention au parenthésage :

B1 (0|1)

B2 0|1

{B1}* et {B2}* : non équivalents (sauf flex).

Partie 1: définition d'états

Idée : appliquer des règles de façon conditionnelle

- L'analyseur lexical possède des états E_1, E_2,...
- A tout moment, il se trouve dans un état.
- Au départ, il est dans l'état nommé `INITIAL`
- Une action `BEGIN` E_i le met dans l'état E_i.
- On peut écrire des règles qui ne s'appliquent que dans un état bien précis.

Définition des états (dans la partie 1)

- %s <nom d'état> : état standard.
- %x <nom d'état> : état exclusif.

Partie 2 : utilisation des états de départ

```
%s etat1
%x etat2
...
%%
```

- Lorsque l'analyseur est dans l'état **etat1**, les seules règles actives sont :
  - celles préfixées par **<etat1>**,
  - celles non préfixées.
- Lorsque l'analyseur est dans l'état etat2, les seules règles actives sont celles préfixées par <etat2>.

Utilisation des états de départ

```
%s etat1
%x etat2
...
%%
<etat1>0*          {action 0}
<etat2>1*          {action 1}
2*                 {action 2}
```

- Dans l'état etat1, les règles 0* et 2* peuvent s'appliquer.
- Dans l'état etat2, seule la règle 1*peut s'appliquer.
- Dans l'état INITIAL, seule la règle 2* peut s'appliquer.

États : exemple

Supprimer les commentaires en C :

```
%x COMMENTAIRE

%%

"/*"                    {BEGIN COMMENTAIRE;}
<COMMENTAIRE>"*/"       {BEGIN INITIAL;}
<COMMENTAIRE>           .|\n  ;
```

Remarque : une chaîne de caractères de la forme `"..../*....*/...."` est modifiée par l'analyseur produit par ce source.

Partie 2 : règles = motifs → actions

- Chaque règle est de la forme :

  expression { <code C> }

- Exemple : transformer toutes les majuscules en minuscules.

  [A-Z] { printf ("%c", yytext[0] + 'a' - 'A'); }

ou mieux :

  [[:upper:]] { printf ("%c", tolower(yytext[0])); }

Partie 2 : application d'une règle

- Lorsque yylex() est appelée, elle cherche dans l'entrée la règle dont l'expression reconnaît le plus long motif (en considérant aussi l'état courant).
- Si plusieurs règles reconnaissent ce plus long motif, la première apparaissant dans le source lex est appliquée.
- Si aucune règle ne s'applique, le caractère suivant de l'entrée est considéré reconnu, et il est recopié sur la sortie.

Le texte reconnu

- Lorsque la chaîne de caractères reconnue est déterminée,
  - elle est sauvegardée dans la variable yytext, sa longueur est sauvée dans yyleng.

yytext[0]	yytext[1]	...	yytext[yyleng-1]	'\0'

- Exemple : texte reconnu : `if`
  - `yyleng` vaut 2
  - `yytext[0]=='i', yytext[1]=='f', yytext[2]=='\0'`

Lex : caractères spéciaux

- `a` : le caractère `'a'`
- \x où x est special : le caractère x lui-même.
- `.` : tout caractère sauf newline
- `[abc]` : soit `'a'`, soit `'b'`, soit `'c'`
- `[b-di-l]`: `'b'`, `'c'`, `'d'`, `'i'`, `'j'` `'k'` ou `'l'`
- `[^a-g]`: tout caractère pas entre `'a'` et `'g'`

e : expression

- e* : un nombre arbitraire de e.
- e+ : au moins une occurrence de e.
- e? : zéro ou une occurrence de e.
- e{2,5} : entre 2 et 5 occurrences de e.
- \x, pour x ¹ a,b,n,r,f,t,v : le caractère x.
- `"?+["` : la chaîne `?+[`

Lex : caractères spéciaux (2)

- {nom}: l'expansion de nom, qui doit avoir été défini
- `(e)`: désigne la même expression que e. Les parenthèses sont utilisées pour le groupement.
- e|e': soit e, soit e'
- e/e' : e suivi d'un motif reconnu par e'.
- e$: e en fin de ligne.
- ^e : e en début de ligne.
- `<etat>`e : une règle d'application conditionnelle.
- `<<EOF>>` : fin de fichier (flex).
- Entre `[]`, les seuls caractères spéciaux sont `\`, `^`, `-`
- Un seul ^ / $ par règle, hors de parenthèses.

Lex : caractères spéciaux (3)

- Classes de caractères : cf. <ctype.h>

[:alpha:]	filtre 1 caractère alphabetique,
[:digit:]	filtre 1 chiffre,
[:alnum:]	([:alpha:] | [:digit:])
[:blank:]	espace ou tabulation. Non portable
[:cntrl:]	caractère de contrôle
[:lower:]	minuscules,
[:upper:]	majuscules,
[:punct:]	ponctuation,
[:print:]	caractère imprimable.
[:graph:]	caractère imprimable sauf ''.
[:space:]	blanc (locale POSIX : '', \t, \v, \f, \n, \r)
[:xdigit:]	chiffre hexadecimal

Exemple : compter lignes et caractères

```
{%
#include <stdio.h>
int num_lignes = 0;
int num_chars = 0;
%}
%%
\n      {num_lignes++; num_char++;}
.       {num_char++;}
%%
int main (void){
    yylex ();
    printf("Le fichier a %d lignes "
           "et %d caractères",
           num_lignes,num_chars);
    exit(0);
}
```

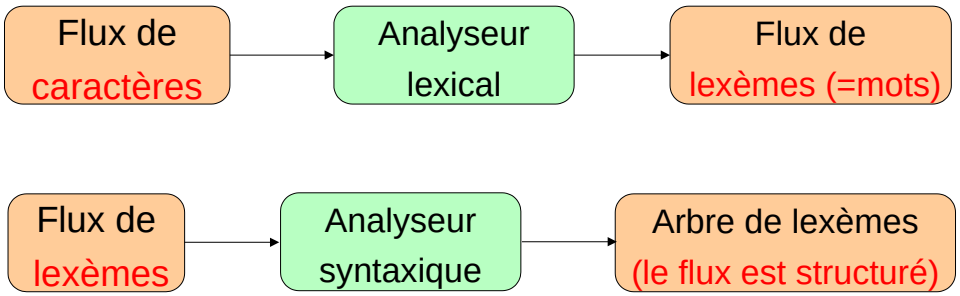
Partie 3 : code utilisateur

- La partie 3 contient du code C.
- lex recopie tout ce code dans le fichier lex.yy.c qu'il génère, après la définition de la fonction d'analyse yylex().

Compléments

- En fin d'entrée, yylex() appelle yywrap()
- Valeur retour de yywrap() :
  - 1: yylex() finit l'analyse (plus d'autre fichier)
  - 0: yylex() continue l'analyse sur le **`FILE *yyin`**
- yymore() demande la concaténation de la chaîne suivante à la fin de ce qui est déjà dans yytext.
- yyless(k) permet de replacer dans le flux d'entrée les caractères reconnus sauf les k derniers.

L'analyse syntaxique



- Remarque sur la terminologie : L'analyseur syntaxique voit les lexèmes comme de simples lettres (et non comme des mots).
- Les chaînes de lexèmes qu'il accepte comme correctes sont appelées des mots (et non des phrases)

Grammaires algébriques

- Moyen de décrire quels sont les flux de lexèmes corrects et comment ils doivent être structurés.
- Chaque lexème est vu comme une lettre.
- La grammaire décrit comment les lettres peuvent être assemblées pour former des mots.
- La grammaire fournit une description des mots (= programmes) syntaxiquement corrects
- Exemple : en C :
  - Entiers, mots-clés, noms = lexèmes
    = lettres de la grammaire.
    Programmes = mots pour la grammaire.

Exemple 1

- Grammaire G décrivant 4 phrases en français :

1) <phrase> → <sujet> <verbe>
2) <sujet> → je
3) <sujet> → tu
4) <verbe> → fais
5) <verbe> → prends

« Lettres » de l'alphabet terminal: "je" "tu" "fais" "prends"

« Lettres » de l'alphabet non-terminal : "<phrase>", "<sujet>", "<verbe>".

La grammaire G a 5 règles.

Phrases françaises décrites (= mots engendrés par G) :

{ je fais, tu fais, je prends, tu prends }

Pourquoi le besoin des grammaires ?

- Expression rationnelle : expression lex utilisant les lettres, la concaténation, le « ou » | et l'étoile *.
- Les expressions rationnelles sont-elles assez puissantes pour décrire des constructions de langages de programmation ?

  Non !
- Exemple : mots sur l'alphabet {},{} bien parenthésés, comme (()()).
- Ce langage ne peut pas être décrit par une expression rationnelle.

D'autres exemples de grammaires algébriques

- Grammaire qui décrit le langage a*b* :

(1) S → ε

(2) S → aS

(3) S → Sb

- Grammaire qui décrit les mots sur l'alphabet {},{} qui sont « bien parenthésés » :

(1) P → ε

(2) P → (P)

(3) P → PP

Autre exemple qui n'est pas rationnel

- Grammaire qui décrit l'ensemble des mots $\{a^n b^n \mid n \geq 0\} = \{\varepsilon, ab, aabb, aaabbb, ...\}$

(1) $S \rightarrow \varepsilon$

(2) $S \rightarrow aSb$

- On montre que le langage $\{a^n b^n \mid n \geq 0\}$ ne peut pas être décrit par une expression rationnelle.
- Inversement, tout langage décrit par une expression rationnelle peut être décrit par une grammaire

A.B	A*	A|B
$S \rightarrow AB$	$S \rightarrow AS$	$S \rightarrow A$
	$S \rightarrow \varepsilon$	$S \rightarrow B$

Expressions arithmétiques

(1) $E \rightarrow E + E$

(2) $E \rightarrow E * E$

(3) $E \rightarrow (E)$

(4) $E \rightarrow ID$

---

(1) $E \rightarrow E + T$

(2) $E \rightarrow T$

(3) $T \rightarrow T * F$

(4) $T \rightarrow F$

(5) $F \rightarrow (E)$

(6) $F \rightarrow ID$

Définitions : mots et langages

- Alphabet $\mathcal{A}$: ensemble fini de lettres.
- $\mathcal{A}^*$ est l'ensemble des mots formés de lettres de $\mathcal{A}$.
- Langage: ensemble (fini ou infini) de mots.
- Longueur |u| d'un mot $u \in \mathcal{A}^*$: nombre de lettres
  - Exemple : |abac| = 4
- ε : mot vide (de longueur nulle) : $|\varepsilon| = 0$.
- Concaténation de deux mots u et v : mot uv, obtenu en collant u et v.
- Exemple : u = aba, v = cab, uv = abacab

Définitions : grammaires algébriques

G = $\{\mathcal{T}, \mathcal{V}, S, \mathcal{R}\}$

- $\mathcal{T}$: alphabet des terminaux
- $\mathcal{V}$: alphabet des variables
- $S \in \mathcal{V}$: symbole de départ ou axiome de la grammaire
- $\mathcal{R}$: ensemble fini de règles. Chaque règle $r \in \mathcal{R}$ est de la forme :

$V \rightarrow \alpha$, où $V \in \mathcal{V}$ et $\alpha \in (\mathcal{T} \cup \mathcal{V})^*$

Comment fonctionne une grammaire

- Une grammaire sert à décrire un langage.
- On part du symbole de départ S.
- Pas de dérivation : $\alpha T \beta \rightarrow \alpha \gamma \beta$ où
  $T \rightarrow \gamma$ est une règle

On note $A \rightarrow^* \alpha$ si

$\alpha = A$ ou $A \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \ldots \alpha$

$(A \in \mathcal{V}, \quad \alpha, \alpha_1, \alpha_2, \ldots \in (\mathcal{V} \cup \mathcal{T})^*)$

- Langage engendré par la grammaire G :
  $L(G) = \{u \in \mathcal{T}^* \mid S \rightarrow^* u\}$
- Langage étendu engendré par la grammaire G :
  $LE(G) = \{u \in (\mathcal{V} \cup \mathcal{T})^* \mid S \rightarrow^* u\}$

Exemple de dérivation et d'arbre

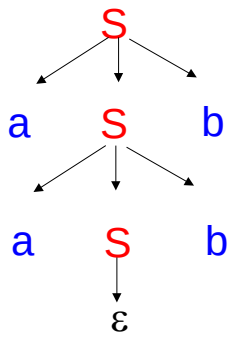
(1) $S \rightarrow \varepsilon$

(2) $S \rightarrow aSb$

$S \rightarrow aSb \rightarrow aaSbb \rightarrow aabb$

Donc $S \rightarrow^* aabb$

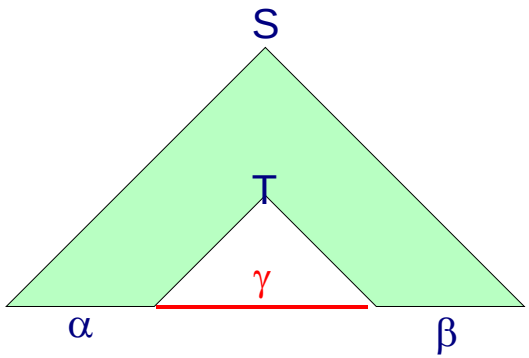
Arbre de dérivation :



Arbre de dérivation

- Arbre de dérivation : il traduit l'application des règles dans une dérivation sans refléter complètement l'ordre d'application des règles

$S \rightarrow^* \alpha T \beta \rightarrow \alpha \gamma \beta$



Arbre de dérivation

- Formellement, un arbre de dérivation est tel que

1) Chaque noeud est étiqueté par un élément de $\mathcal{V} \cup \mathcal{T} \cup \{\varepsilon\}$
2) La racine est étiquetée par S.
3) Les feuilles sont étiquetées par des éléments de $\mathcal{T}$ ou par ε.
4) Les noeuds étiquetés par ε sont fils uniques.
5) Si un noeud est étiqueté par $A \in \mathcal{V}$ et les fils sont étiquetés par $u_1, u_2, \ldots u_k \in \mathcal{V} \cup \mathcal{T}$, (dans cet ordre), alors

$A \rightarrow u_1 u_2 \ldots u_k$ est une règle

Grammaires ambiguës

- Une grammaire est ambiguë si un mot possède deux arbres de dérivation différents.
- Exemple : mots « bien parenthésés » :

(1) $P \rightarrow \varepsilon$

(2) $P \rightarrow (P)$

(3) $P \rightarrow PP$

- Deux arbres pour le mot ε : grammaire ambiguë

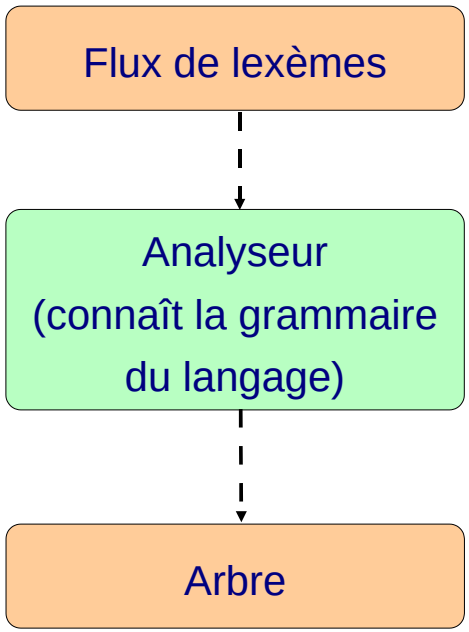
Analyse syntaxique

- Une grammaire G étant fixée, le problème de l'analyse syntaxique est le suivant :
- Donnée :
  - Un mot (= un programme) sur l'alphabet des symboles terminaux.
- Questions :
  - La grammaire G engendre-t-elle ce mot ?
  - Si oui, comment trouver efficacement un arbre de dérivation pour ce mot ?

L'analyse syntaxique

- L'analyseur, connaissant la grammaire G :
  - reçoit en entrée un mot u sur l'alphabet terminal.
  - produit en sortie l'arbre de dérivation de u si le mot est engendré par G.
  - indique une erreur syntaxique sinon.

Principe de l'analyse syntaxique



Restrictions

- On suppose que pour toutes les grammaires que l'on considérera :
  - Tous les non-terminaux sont atteignables :

    $\forall Y \in \mathcal{V}$, on a $S \rightarrow^* \alpha Y \beta$
  - Chaque symbole non-terminal peut se dériver en un mot de terminaux :

    $\forall Y \in \mathcal{V}$, on a $Y \rightarrow^* u$ pour $u \in \mathcal{T}^*$
- Toute grammaire peut être transformée pour répondre à ces deux critères sans changer le langage engendré.

Une stratégie d'analyse de haut en bas

- L'analyseur part du symbole de départ de la grammaire et construit l'arbre du haut vers le bas.
- Il « regarde » à chaque étape un seul symbole du flux d'entrée, le symbole d'avance (ou courant).
- Il « regarde » à chaque étape un seul symbole dans l'arbre construit, le plus à gauche non traité.

Analyseurs LL(1)

- Etant donnée une grammaire G,on ne peut pas toujours construire un analyseur LL(1) pour G.
- G doit satisfaire des conditions, en particulier être non-ambiguë.
- Un analyseur LL(1) est déterministe (il ne prend pas de décision au hasard).
- Il n'utilise pas de *backtracking*.
- L'algorithme de l'analyseur LL(1) se base sur une table qui dit quoi faire connaissant
  - le symbole d'avance,
  - le premier symbole non traité dans l'arbre construit

Exemple

$S \rightarrow a\ S\ e$

$S \rightarrow B\ C$

$B \rightarrow b\ B\ e$

$B \rightarrow C$

$C \rightarrow c\ C\ e$

$C \rightarrow d$

$C \rightarrow \varepsilon$

Le mot abdee est-il syntaxiquement correct ?

Si oui, peut-on trouver un arbre de dérivation ?

La table de l'analyseur

L'analyseur se base sur la table suivante.

On verra plus tard comment il l'a construite.

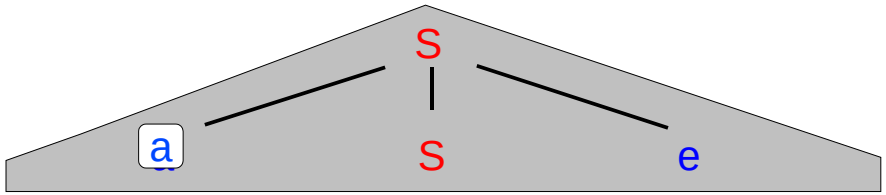
	S	B	C
a	S → a S e	Err	Err
b	S → BC	B → b B e	Err
c	S → BC	B → C	C → c C e
d	S → BC	B → C	C → d

Analyse LL(1) : exemple

S

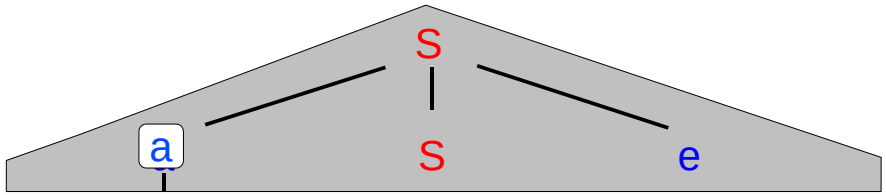
a b d e e

Analyse LL(1) : exemple



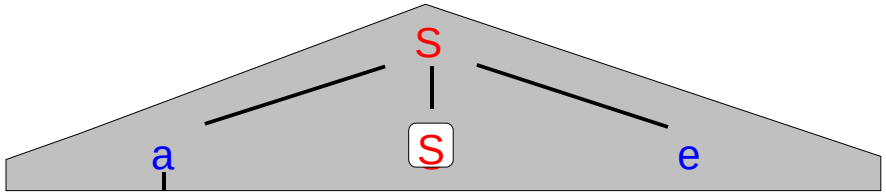
a b d e e

Analyse LL(1) : exemple

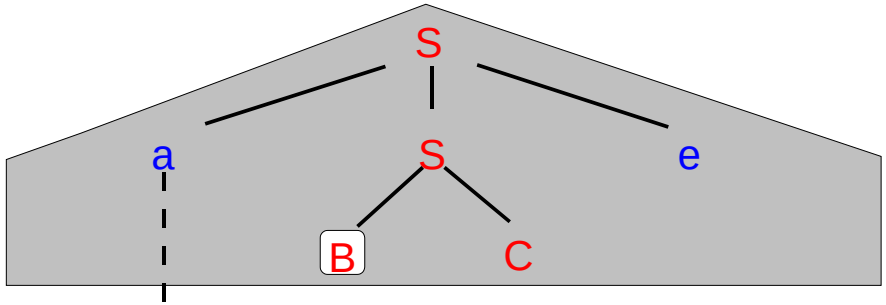


a b d e e

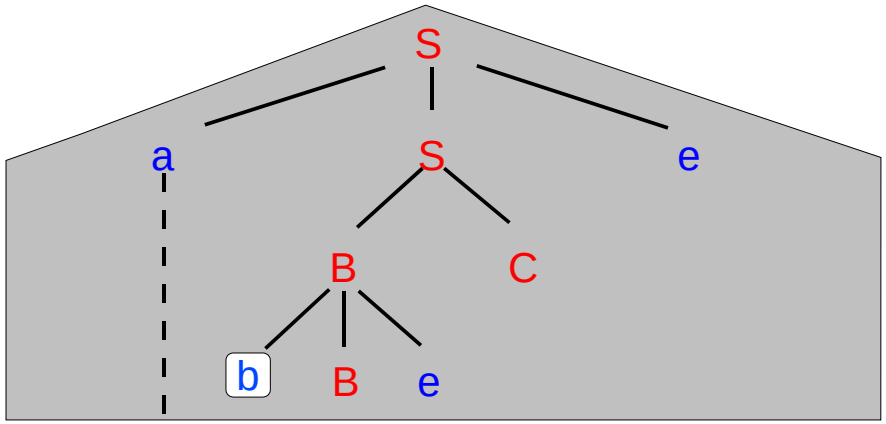
Analyse LL(1) : exemple



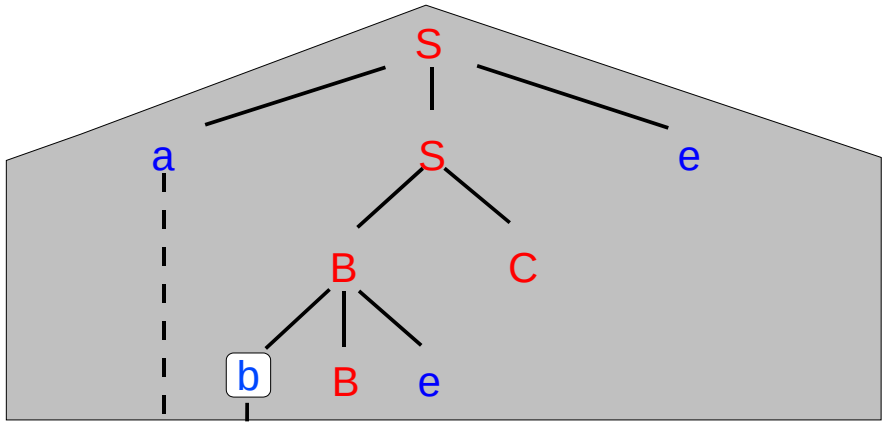
Analyse LL(1) : exemple



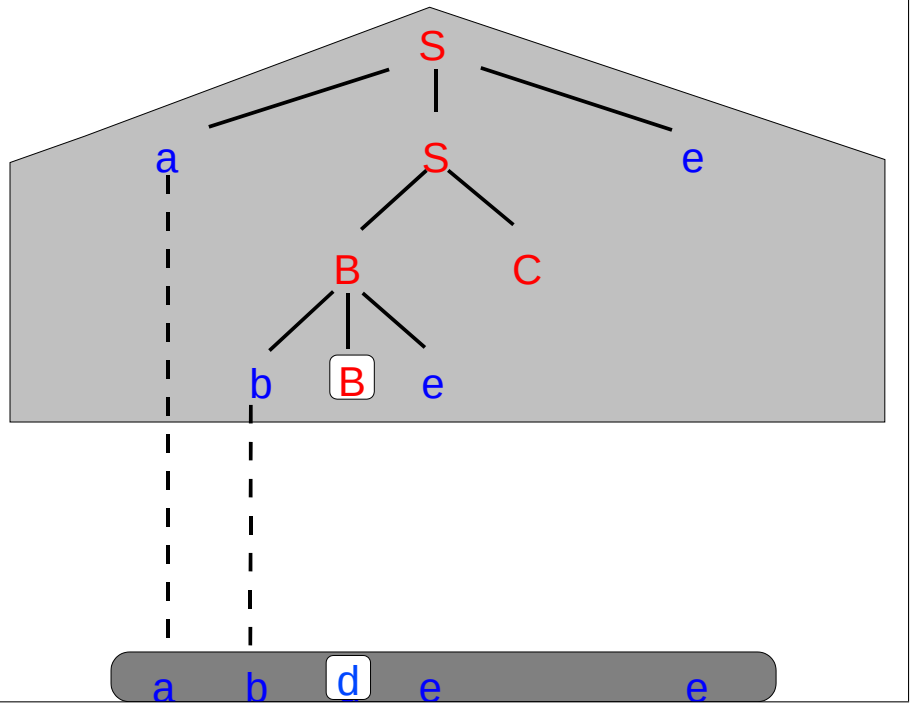
Analyse LL(1) : exemple



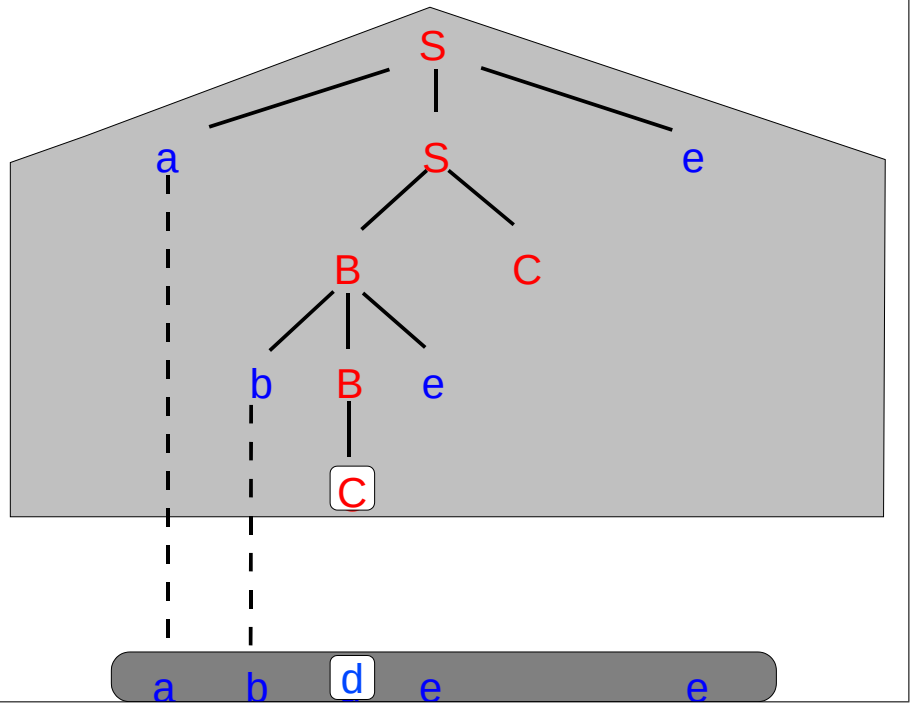
Analyse LL(1) : exemple



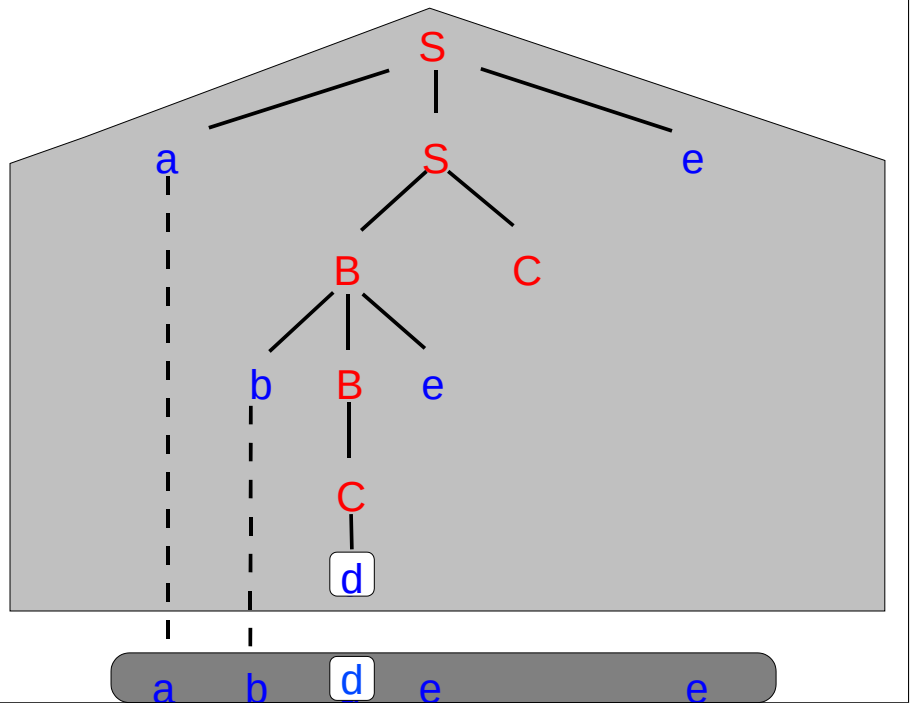
Analyse LL(1) : exemple



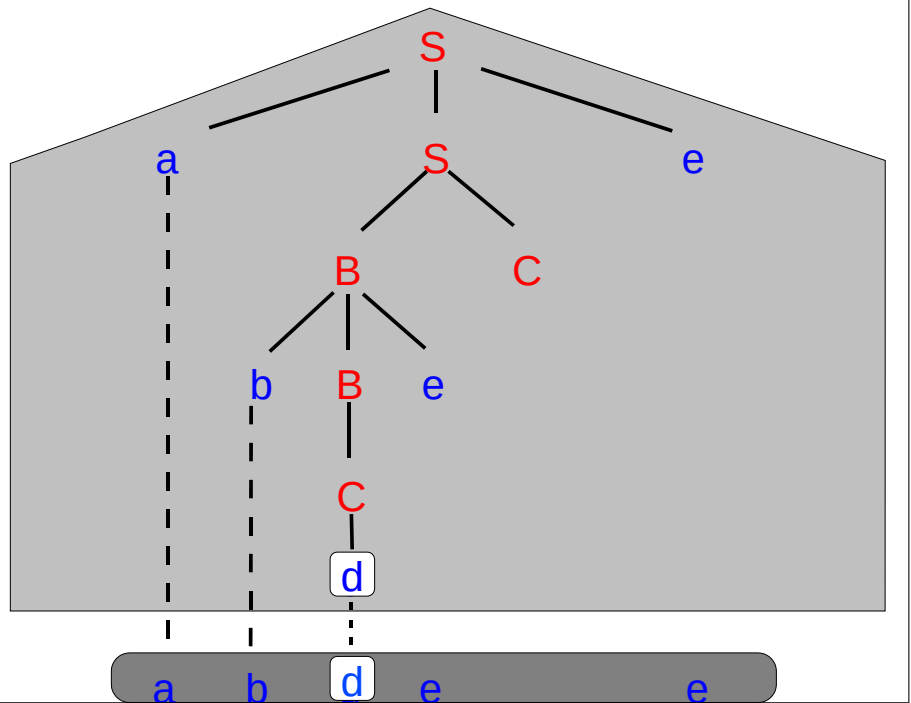
Analyse LL(1) : exemple



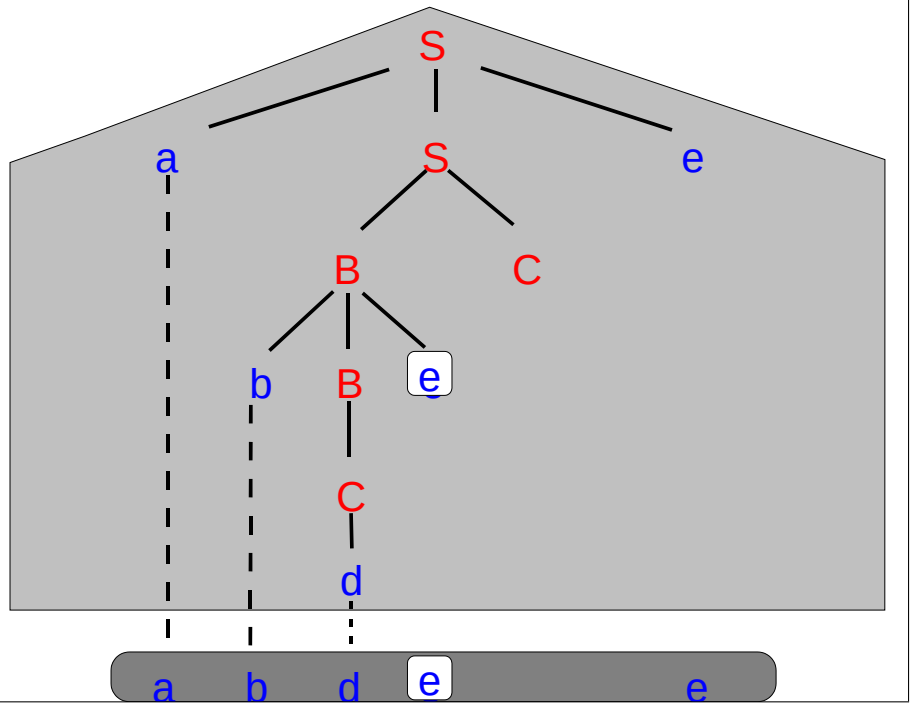
Analyse LL(1) : exemple



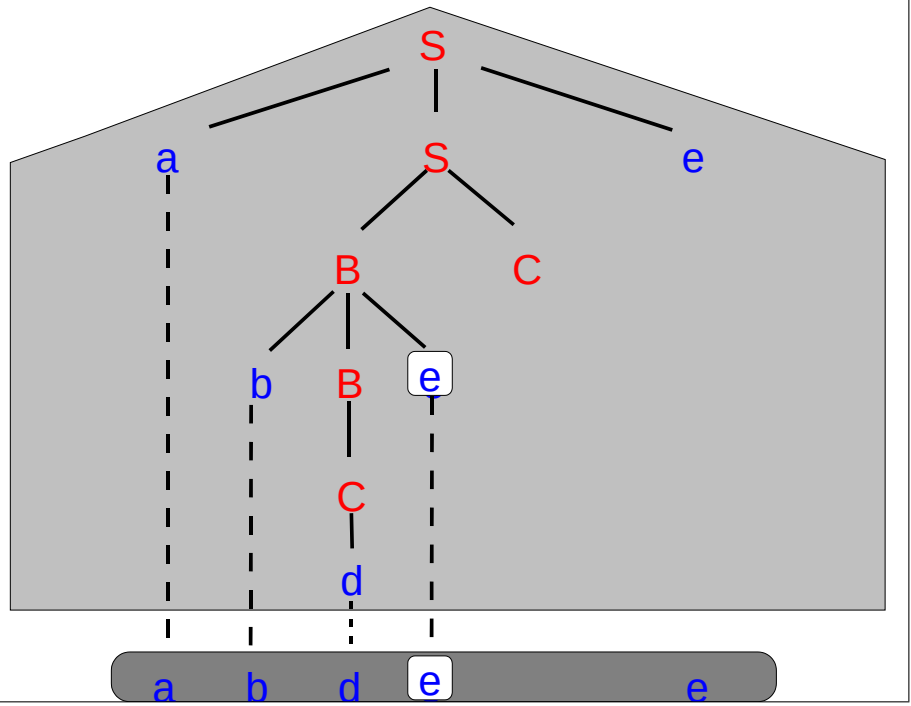
Analyse LL(1) : exemple



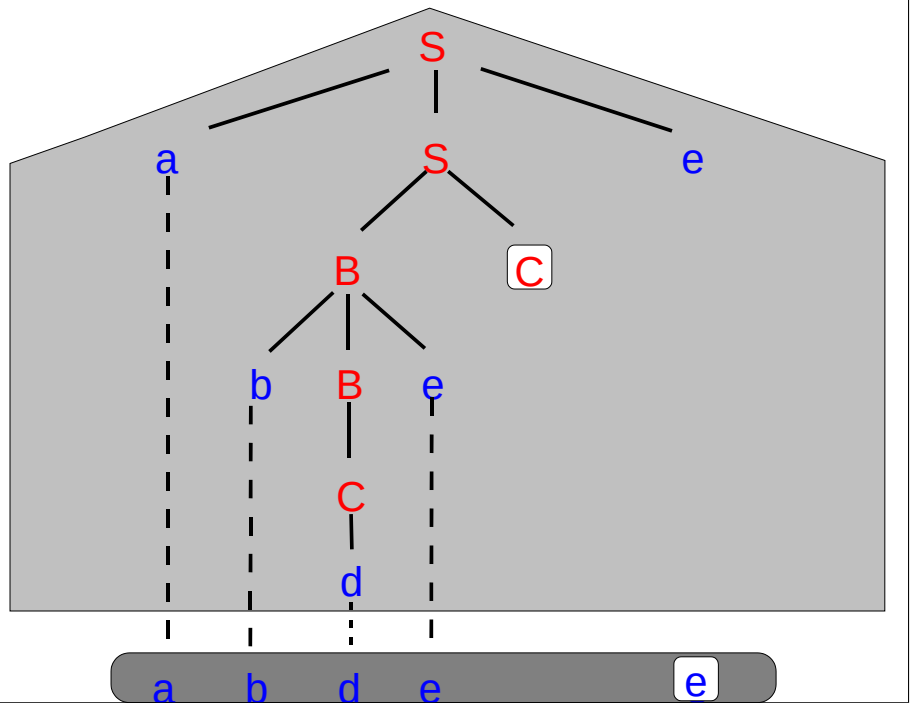
Analyse LL(1) : exemple



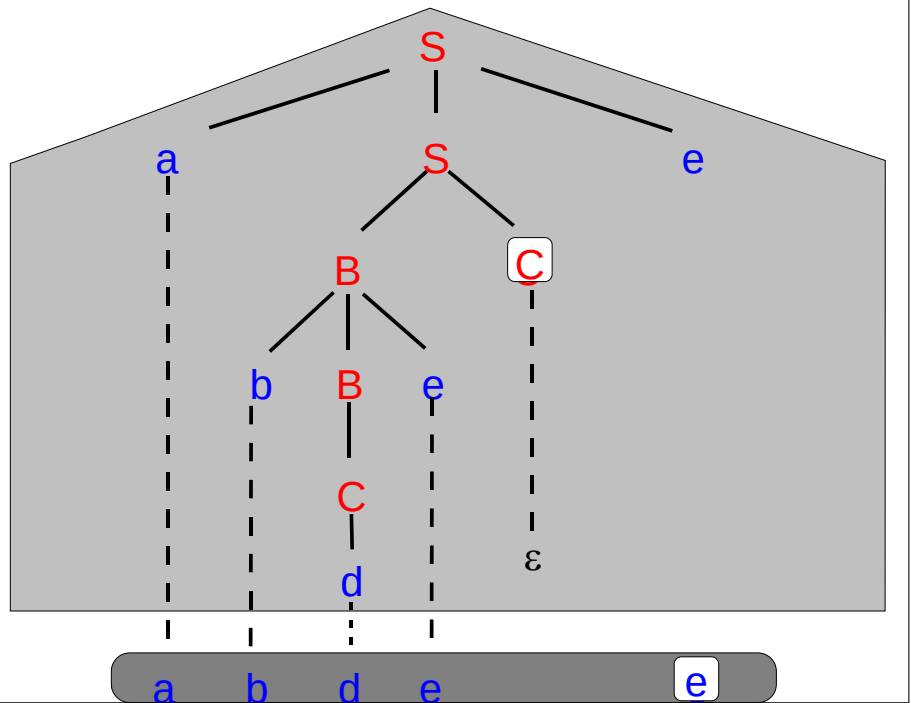
Analyse LL(1) : exemple



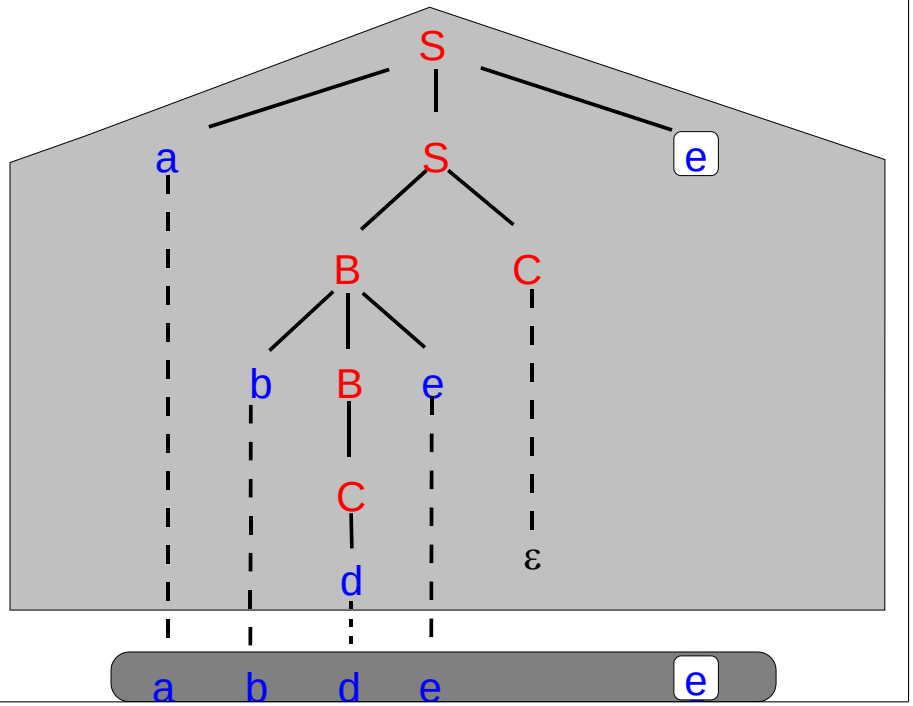
Analyse LL(1) : exemple



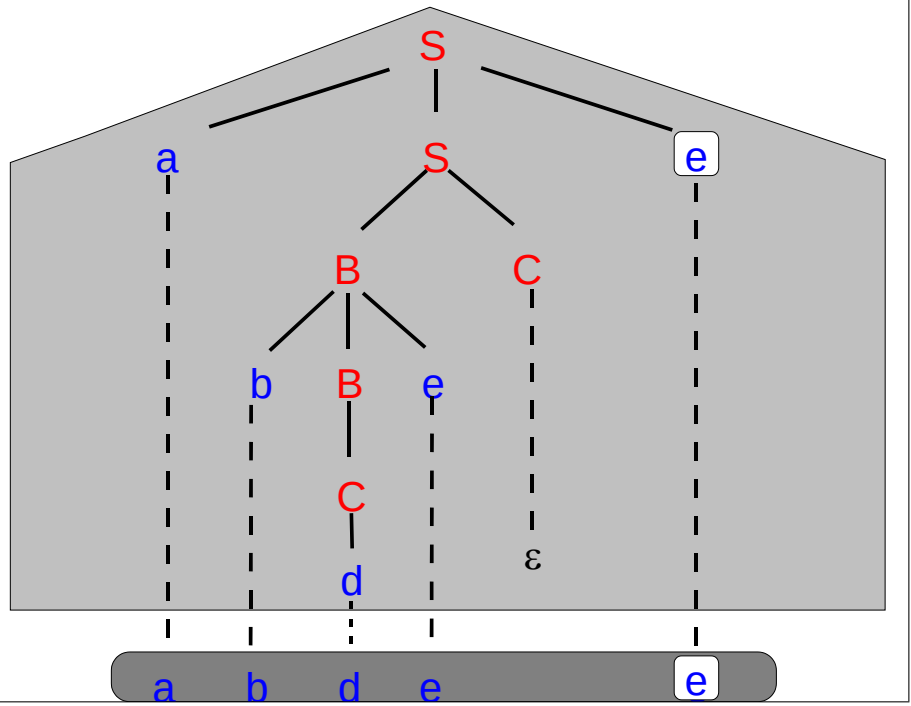
Analyse LL(1) : exemple



Analyse LL(1) : exemple



Analyse LL(1) : exemple



Une grammaire pour laquelle cette stratégie ne marche pas

(1) S → a S b

(2) S → a b

S

?

a b

Les ensembles Premier, Suivant

- Pour prendre une décision, l'analyseur se base sur les symboles qui peuvent apparaître en première position à partir d'une dérivation partant de chaque symbole T.

$T \rightarrow^* a\alpha \qquad \alpha \in (\mathcal{V} \cup \mathcal{T})^*$

$\Leftrightarrow$ a $\in$ Premier(T)

- Il doit aussi calculer les symboles terminaux qui peuvent suivre un non-terminal T dans une dérivation à partir du symbole de départ S.

$S \rightarrow^* \alpha T a \beta \qquad \alpha,\beta \in (\mathcal{V} \cup \mathcal{T})^*$

$\Leftrightarrow$ a $\in$ Suivant(T)

Exemple

S → a S e
S → BC
B → b B e
B → C
C → c C e
C → d
C → ε

Premier(S) = {a,b,c,d}
Premier(B) = {b,c,d}
Premier(C) = {c,d}

Suivant(S) = {e}
Suivant(B) = {c,d,e}
Suivant(C) = {c,d,e}

Calcul des effaçables

Un non-terminal Y est effaçable si $Y \rightarrow^* \varepsilon$.

$E(0) = \{Y \in \mathcal{V} \mid Y \rightarrow \varepsilon.\}$

$E(i+1) = E(i) \cup \{Y \in \mathcal{V} \mid Y \rightarrow \alpha \in E(i)^*\}$

$E(0) \subset E(1) \subset \ ... \ \subset E(p) = E(p+1) = ... \ \subset \mathcal{V}$

Proposition : E(p) = ensemble des effaçables

Algorithme de calcul des ensembles Premier(Y), $Y \in \mathcal{V}$

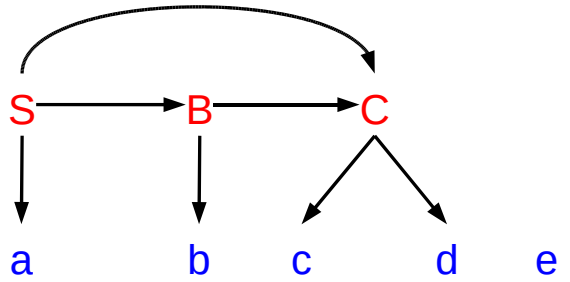
- Pour chaque règle qui peut s'écrire sous la forme
  $Y \rightarrow \alpha a \beta \qquad (a \in \mathcal{T}, \alpha,\beta \in (\mathcal{V} \cup \mathcal{T})^*)$
  telle que $\alpha \rightarrow^* \varepsilon$, ajouter a à Premier(Y).

- Pour chaque règle qui peut s'écrire sous la forme
  $Y \rightarrow \alpha X \beta \qquad (X \in \mathcal{V}, \alpha,\beta \in (\mathcal{V} \cup \mathcal{T})^*)$
  telle que $\alpha \rightarrow^* \varepsilon$, ajouter Premier(X) à Premier(Y).

- Représentation sous forme de graphe.

Graphe des Premiers de la grammaire de l'exemple

S → a S e
S → BC
B → b B e
B → C
C → c C e
C → d
C → ε



S, B et C sont effaçables.

Généralisation du calcul de Premier(X)

- On peut calculer, pour $\beta \in (\mathcal{V} \cup \mathcal{T})^*$

  $$\text{Premier}(\beta) = \{a \in \mathcal{T} \mid \beta \rightarrow^* a\alpha\}$$

- Par convention, si $\beta = b \in \mathcal{T}$, Premier(b) = {b}.

- Si $\beta = X \in \mathcal{V}$, Premier(X) est calculé par l'algorithme précédent.

- Si $\beta = y_1 \ldots y_k$, avec $y_i \in \mathcal{V} \cup \mathcal{T}$,

  $\text{Premier}(\beta) = \text{Premier}(y_1)$ si y_1 non effaçable

  $= \text{Premier}(y_1) \cup \text{Premier}(y_2 \ldots y_k)$ sinon

Algorithme de calcul des ensembles Suivant(Y), $Y \in \mathcal{V}$

- Pour chaque règle de la forme :

  $X \rightarrow \alpha Y \beta$ $\quad (\alpha,\beta \in (\mathcal{V} \cup \mathcal{T})^*)$

  ajouter Premier(β) à Suivant(Y).

- Pour chaque règle qui peut s'écrire sous la forme

  $X \rightarrow \alpha Y \beta$ $\quad (X \in \mathcal{V}, \alpha,\beta \in (\mathcal{V} \cup \mathcal{T})^*)$

  telle que $\beta \rightarrow^* \varepsilon$, ajouter Suivant(X) à Suivant(Y).

$$S \rightarrow^* \gamma\, X a\, \delta \rightarrow \gamma\, \alpha Y \beta\, a\, \delta \rightarrow^* \gamma\, \alpha Y a\, \delta$$

Graphe des Suivants de la grammaire de l'exemple

$S \rightarrow a\ S\ e$

$S \rightarrow B\ C$

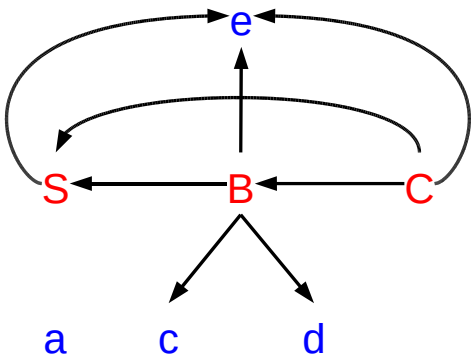
$B \rightarrow b\ B\ e$

$B \rightarrow C$

$C \rightarrow c\ C\ e$

$C \rightarrow d$

$C \rightarrow \varepsilon$



S, B et C sont effaçables

La table de prédiction

X : symbole à dériver dans l'arbre (non-terminal)

a : symbole d'avance (terminal).

On peut appliquer $X \rightarrow \alpha$ quand :

$a \in \text{Premier}(\alpha)$

ou

α est effaçable et $a \in$ Suivant(X)

On dit que la règle

$$X \rightarrow \alpha$$

est prédite sur le couple (X,a)

Quand une grammaire est-elle LL(1) ?

- Une grammaire est LL(1) lorsque, pour chaque couple (X,a) $\in \mathcal{V} \times \mathcal{T}$, il y a au plus une règle prédite par l'analyseur LL(1).
- Ceci se traduit sur la table d'analyse par : chaque case contient au plus une règle.
- Si une grammaire n'est pas LL(1), il est possible qu'il existe une autre grammaire engendrant le même langage qui est LL(1).

Grammaires non LL(1)

- Une grammaire ambiguë n'est jamais LL(1).
- Une grammaire est récursive gauche lorsqu'elle a une règle utile de la forme :

$$X \rightarrow X\ \alpha$$

Une telle grammaire n'est pas LL(1).

- Une grammaire est non factorisée lorsqu'elle a deux règles (utiles, distinctes) de la forme

$$X \rightarrow \alpha\ \beta$$
$$X \rightarrow \alpha\ \gamma$$

avec Premier(α) $\neq \emptyset$.
Une telle grammaire n'est pas LL(1).

Remarques

- Etant donnée une grammaire G, on peut construire une grammaire G' qui engendre le même langage
  - qui est factorisée
  - qui n'est pas récursive gauche
- Ces deux conditions ne garantissent cependant pas que la grammaire G' est LL(1).

Exemple : expressions arithmétiques

(1) E → E+T
(2) E → T
(3) T → T*F
(4) T → F
(5) F → (E)
(6) F → x

Cette grammaire n'est pas LL(1) : elle est récursive gauche : règles (1) et (3).

Exemple : expressions arithmétiques

(1) E → T E'

(2) E' → + T E'

(3) E' → ε

(4) T → F T'

(5) T' → *F T'

(6) T' → ε

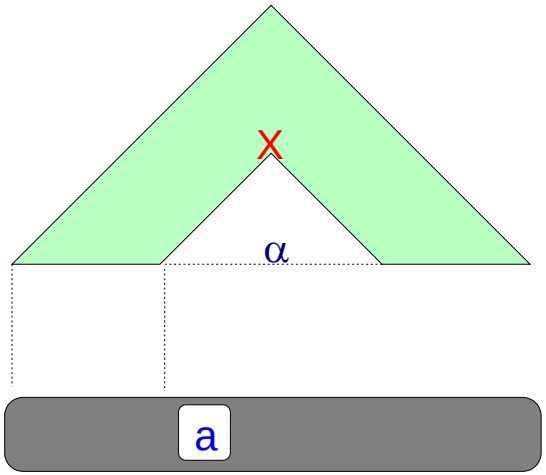
(7) F → x

(8) F → (E)

Cette grammaire

- est LL(1),
- engendre le même langage que la précédente.

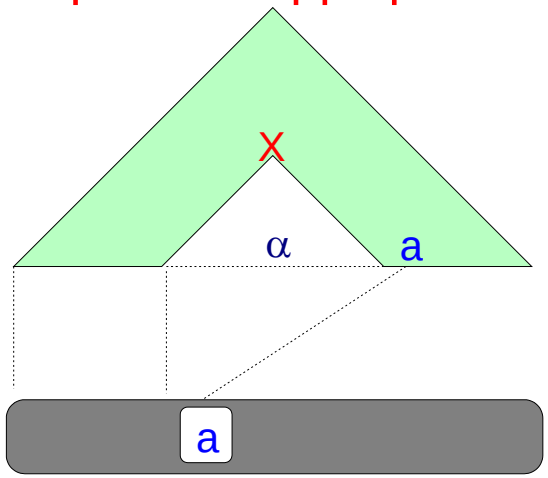
Quand peut-on appliquer X → α ?



Premier cas :

On peut appliquer X → α lorsque a ∈Premier (α)

Quand peut-on appliquer X → α ?



Deuxième cas :

On peut appliquer X → α lorsque

α →* ε et

a ∈Suivant (α)

Exemple : expressions arithmétiques

(1) E → T E'

(2) E' → + T E'

(3) E' → ε

(4) T → F T'

(5) T' → *F T'

(6) T' → ε

(7) F → x

(8) F → (E)

Premier

E	E'	T	T'	F
(, x	+	(, x	*	(, x

Suivant

E	E'	T	T'	F
)	)	+,)	+,)	*, + ,)

Exemple : expressions arithmétiques

Table d'analyse :

	E	E'	T	T'	F
+		E' → + T E'		T' → ε	
*				T' → *F T'	
x	E → T E'		T → F T'		F → x
(	E → T E'		T → F T'		F → (E)
)		E' → ε		T' → ε	

Écrire un analyseur descendant

- On écrit une fonction par non-terminal.
- Chaque fonction est chargée de construire le sous arbre partant du non-terminal correspondant.
- L'analyseur maintient une variable globale pour conserver le symbole d'avance.
- Il doit enfin s'assurer que la fin du source est atteinte lorsque l'arbre est terminé
  ⇒ ajout d'une règle S' → S $, où
  - S', nouveau non-terminal, devient le symbole de départ,
  - $, nouveau terminal, représente la fin de fichier

Exemple : la fonction `F()`

```
void F(void){
  switch (symboleAvance){
    case 'x':
      consommer('x');
      break;
    case '(':
      consommer('(');
      E();
      consommer(')');
      break;
    default :
      erreurSyntaxique();
      break;
   }
}
```

Analyse syntaxique montante

- Idée : construire l'arbre de bas en haut, en partant de la chaîne de terminaux.
- Objectifs :
  - à nouveau, regarder peu de symboles à la fois (idéalement 1) dans le flux d'entrée ;
  - obtenir un algorithme déterministe ;
  - pas de *backtraking*.

Exemple

(1) E → E+T

(2) E → T

(3) T → T∗F

(4) T → F

(5) F → (E)

(6) F → x

On veut analyser : x ∗ (x + x)

Exemple

Pile

Empiler

x ∗ (x + x)

Exemple

Pile

Réduire

F → x

x

x ∗ (x + x)

Exemple

Pile

Réduire

T → F

F

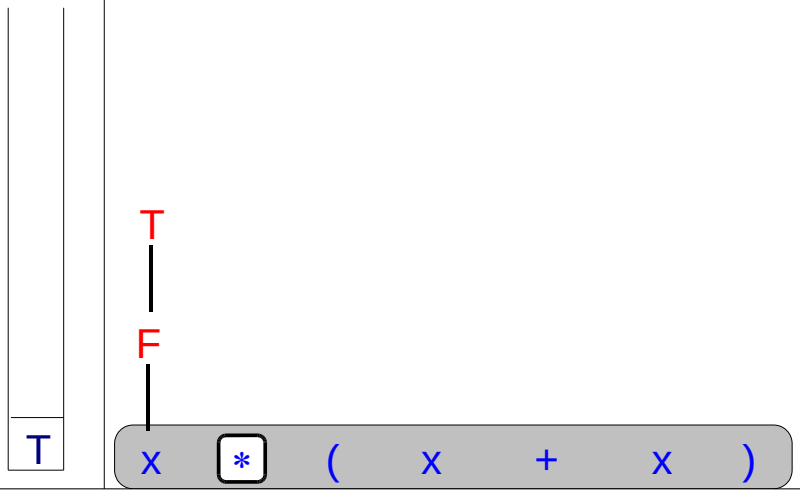
F

x ∗ (x + x)

Exemple

Pile

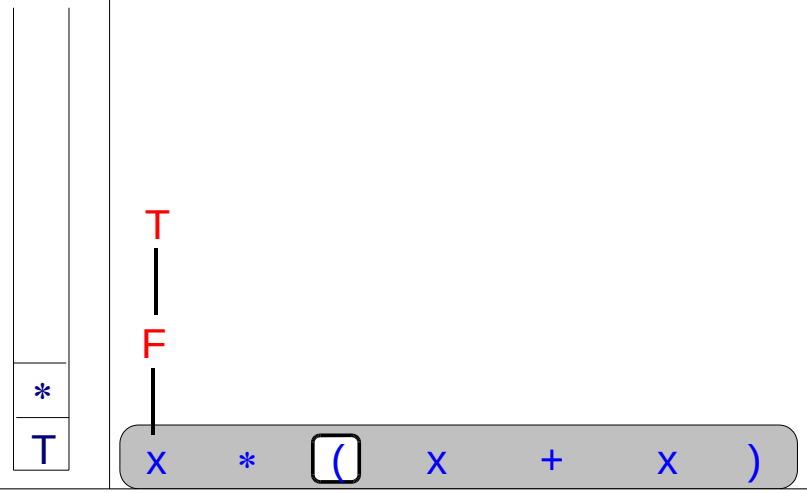
Empiler



Exemple

Pile

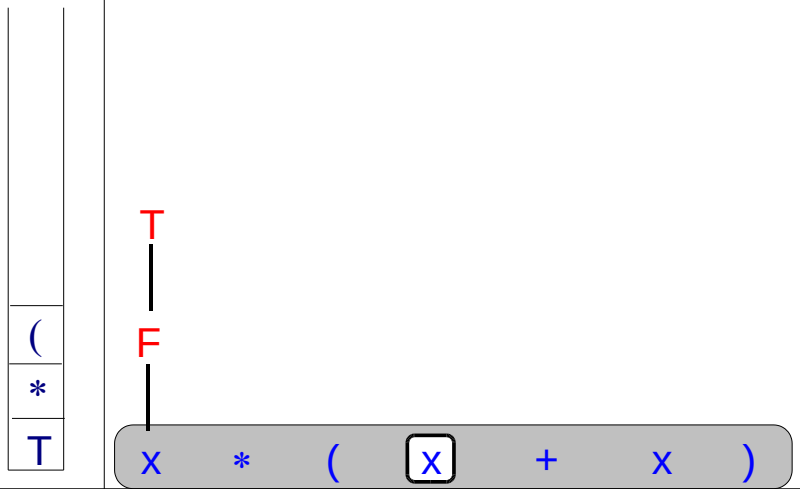
Empiler



Exemple

Pile

Empiler

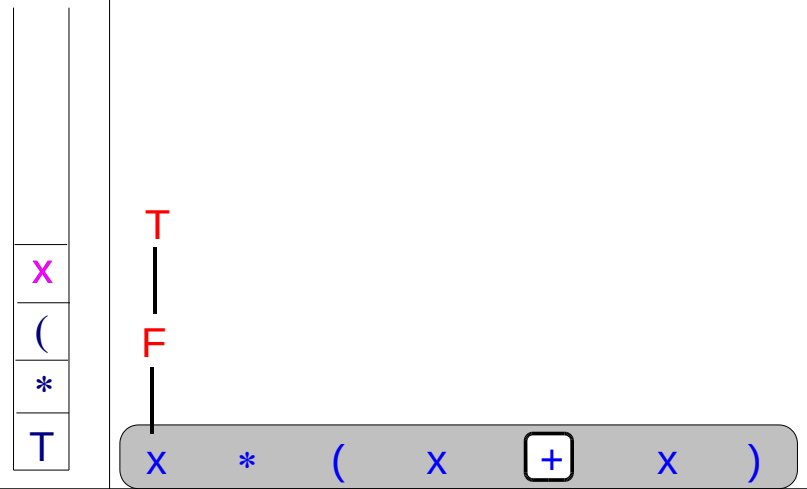


Exemple

Pile

Réduire

F → x

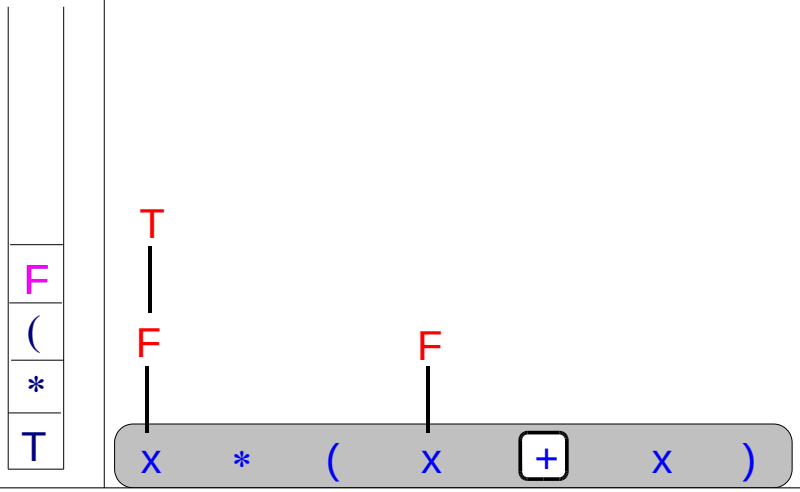


Exemple

Pile

Réduire

$T \rightarrow F$

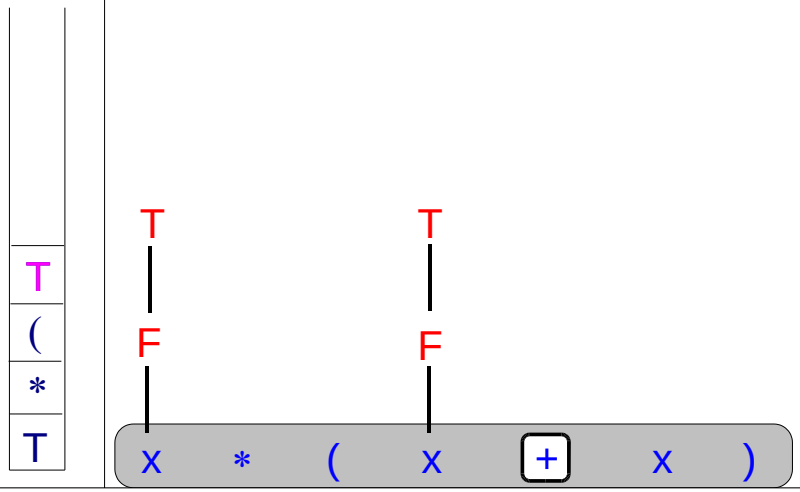


Exemple

Pile

Réduire

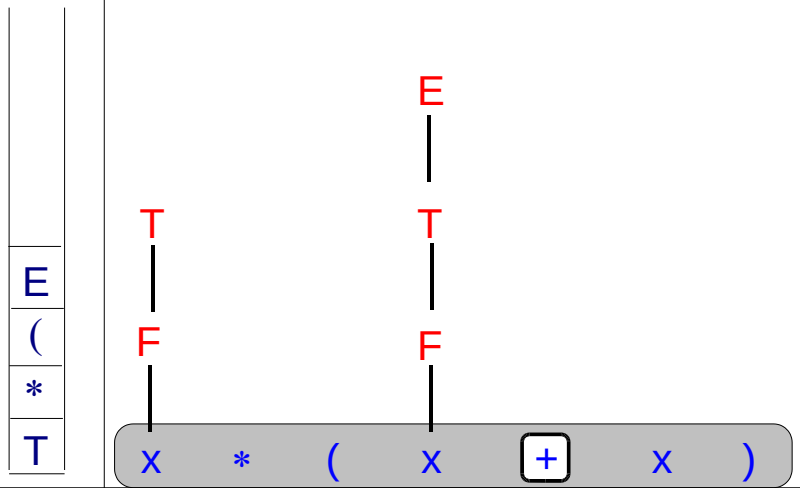
$E \rightarrow T$



Exemple

Pile

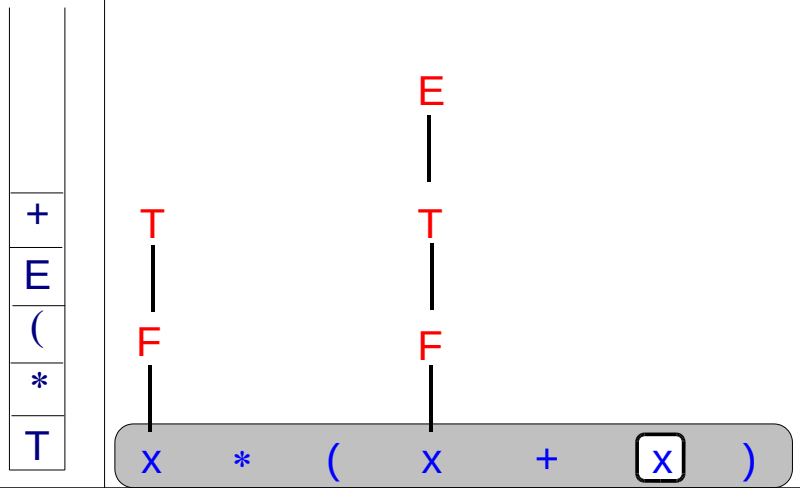
Empiler



Exemple

Pile

Empiler

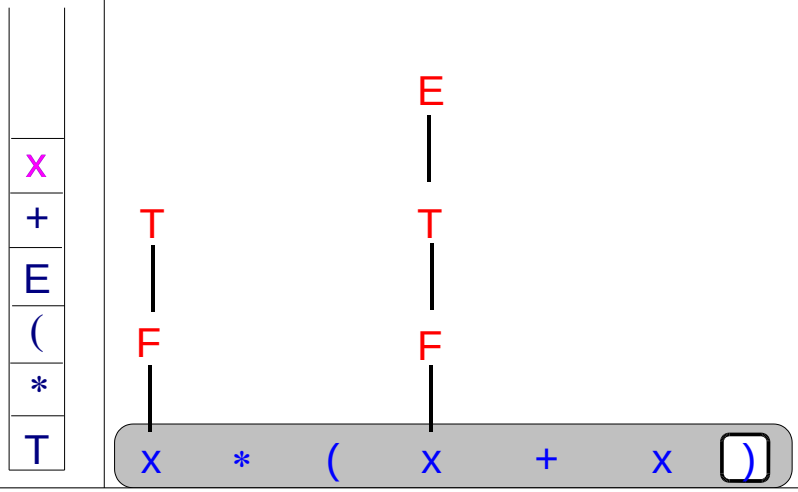


Exemple

Pile

Réduire

$F \rightarrow x$

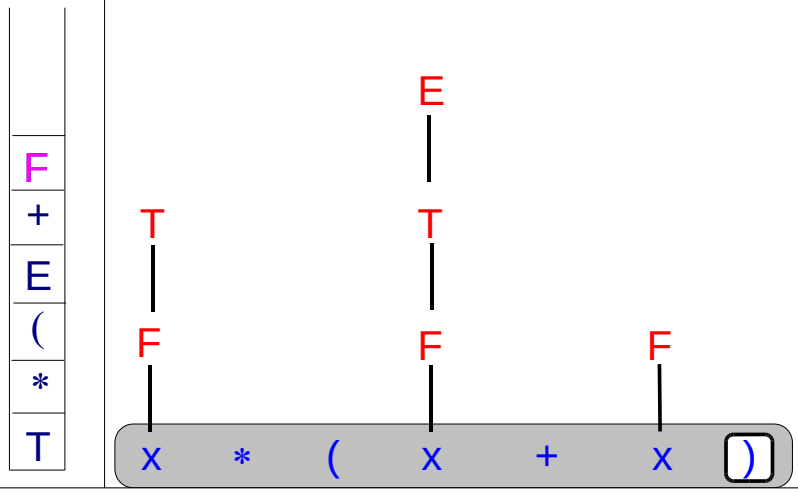


Exemple

Pile

Réduire

$T \rightarrow F$

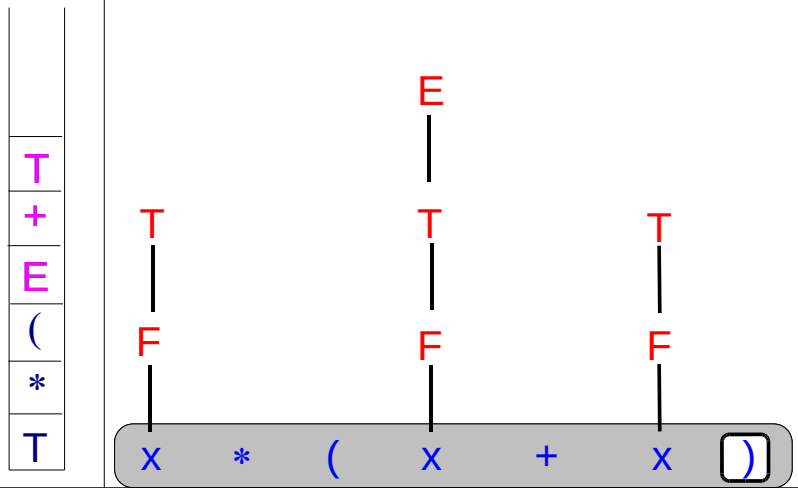


Exemple

Pile

Réduire

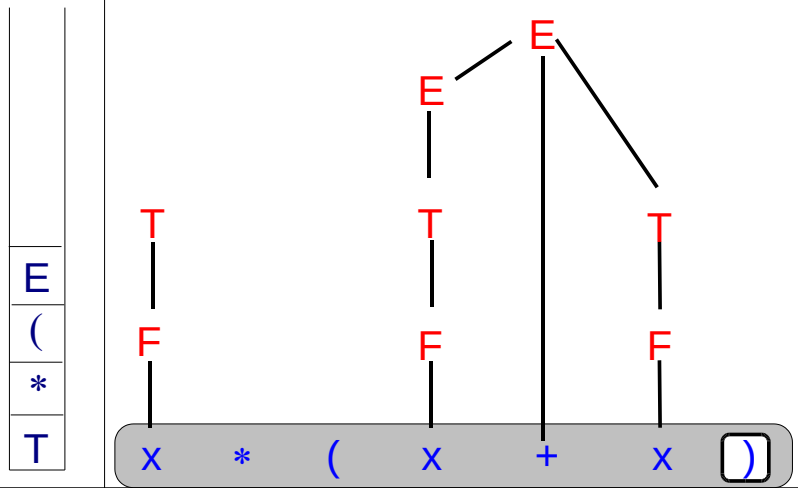
$E \rightarrow E+T$



Exemple

Pile

Empiler

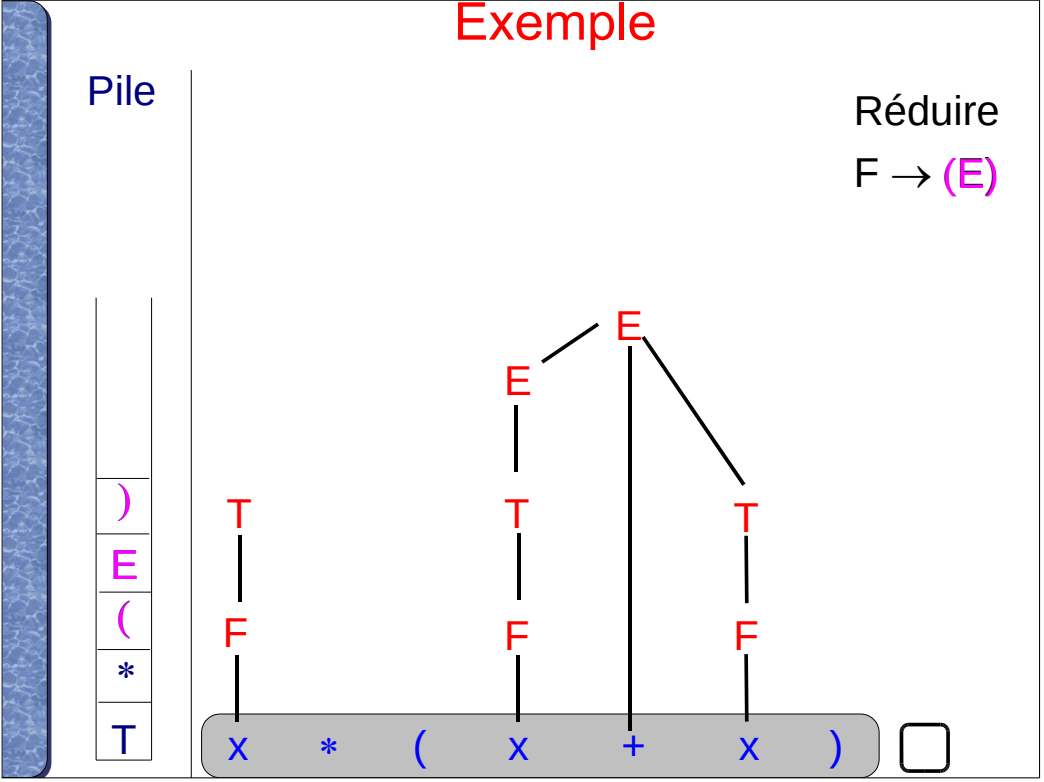


Exemple

Pile

Réduire

F → (E)

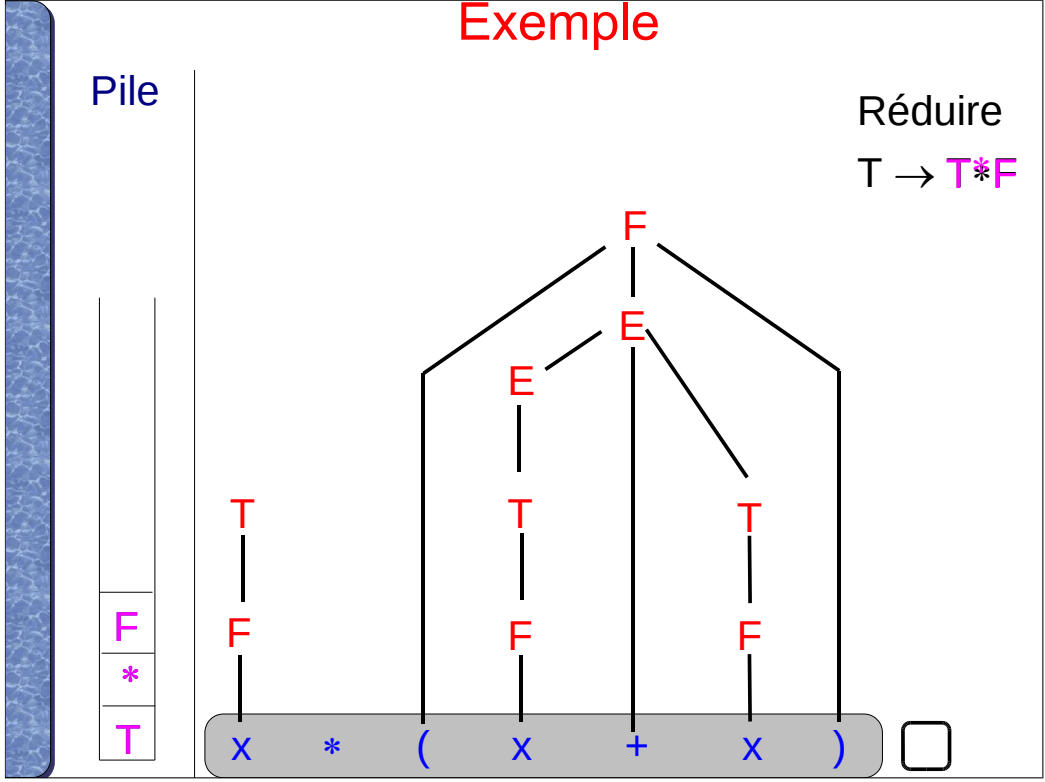


Exemple

Pile

Réduire

T → T*F

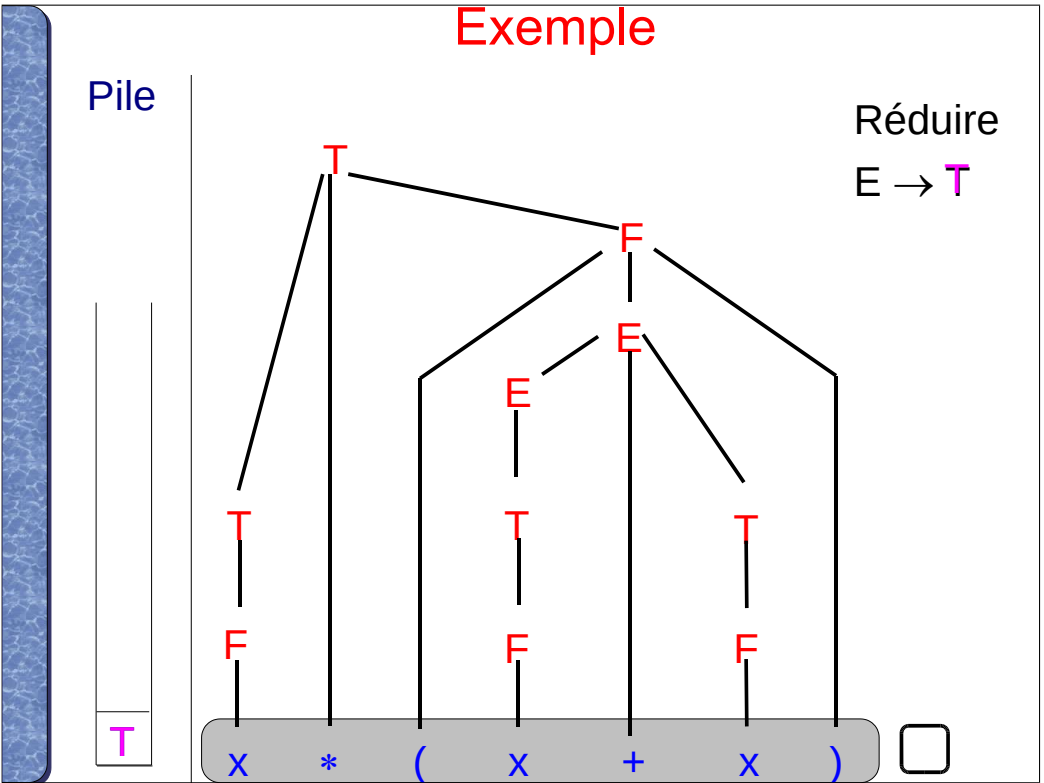


Exemple

Pile

Réduire

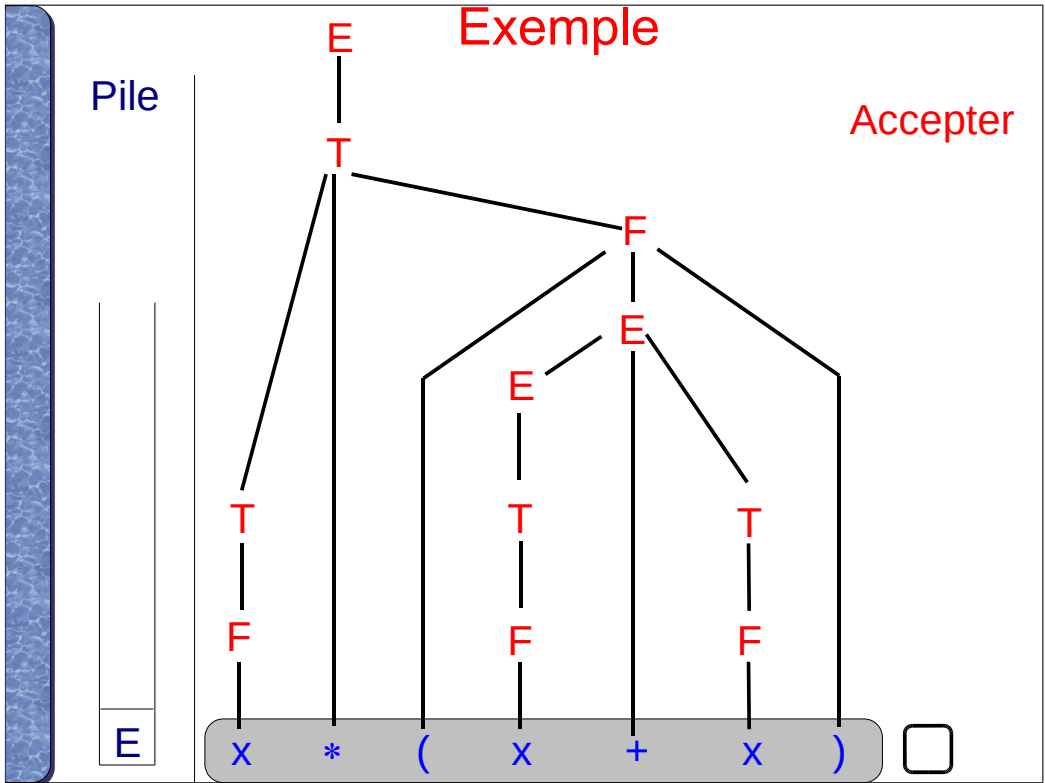
E → T



Exemple

Pile

Accepter



Deux types de décision

- Empilement : consiste à mémoriser le terminal lu.

  On utilise pour cela une pile

- Réduction : consiste à « remonter » dans l'arbre

  Lorsqu'on réduit, on modifie la pile, sans consommer de symbole dans le flux d'entrée.

Un premier type d'analyseur montant : l'analyse LR(0)

- Idée : l'analyseur mémorise
  - tout ce qui a été empilé (grâce à une pile !)
  - dans un emplacement de la pile est mémorisé
    - ce qui a été empilé récemment,
    - ce qu'on doit arriver à empiler dans le futur pour parvenir à une analyse correcte
- Pour mémoriser cela, on utilise des règles marquées, ou items.

Les règles marquées (ou items)

- Si une règle peut s'écrire

  $X \rightarrow \alpha\beta$,

  alors

  $X \rightarrow \alpha \bullet \beta$

  est un item.

- Exemples :
- La règle $E \rightarrow E+T$ fournit les items

  $E \rightarrow \bullet E + T$, $E \rightarrow E \bullet + T$, $E \rightarrow E + \bullet T$, $E \rightarrow E + T \bullet$
- La règle $A \rightarrow \varepsilon$ fournit un seul item : $A \rightarrow \bullet$

Le principe de l'analyse LR(0)

- L'analyseur a une pile d'états pour mémoriser
  - tous les symboles (terminaux ou non) empilés sur la pile des symboles.
  - Dans la case du haut de la pile d'états, on mémorise :
    - les symboles empilés récemment sur la pile des symboles,
    - les symboles que l'on doit arriver à empiler dans le futur pour parvenir à une réduction.

Ces infomations sont codées via des items.

États de l'analyseur

- L'analyseur gère ainsi 2 piles fonctionnant en //
  - La pile des états,
  - La pile des symboles.
- Remarque : La pile des états contient strictement plus d'information que la pile des symboles.
- Sur la pile d'états, il empile des ensembles d'items
- Un ensemble d'items est appelé un état.
- On dit que l'analyseur se trouve dans l'état du sommet de cette pile.

Signification intuitive des états

- Lorsqu'un item $X \rightarrow \alpha \bullet \beta$ se trouve dans l'état du haut de la pile, il code les informations suivantes :
  - α est en haut de la pile des symboles,
  - pour avoir tout le membre droit $\alpha\beta$ de $X \rightarrow \alpha\beta$ en haut de la pile des symboles, il reste à empiler les symboles de β.
- Un état mémorise ainsi toutes les possibilités d'empilement ou de réduction envisageables.

Exemple

$E \rightarrow E + T \mid T$

$E \rightarrow (E) \mid x$

- Si l'analyseur est dans un état contenant l'item

$$E \rightarrow E + \bullet T$$

c'est qu'il y a

$$E +$$

en haut de la pile des symboles.

- S'il se touve dans un état contenant l'item

$$T \rightarrow (E) \bullet$$

c'est qu'il y a

$$(E)$$

en haut de la pile des symboleset que la réduction par $T \rightarrow (E)$ est possible.

Une règle pour marquer la fin d'entrée

- On suppose que l'on a augmenté la grammaire avec une règle

$$S' \rightarrow S \$$$

- où
  - $ est un nouveau terminal,
  - S est l'ancien état de départ,
  - S' est le nouvel état de départ.

Ceci ne change le langage reconnu qu'en « marquant » les mots par $ à la fin.

Clôture directe d'un ensemble d'items

- Soit $\mathcal{E}$ un ensemble d'items.
- La clôture directe de $\mathcal{E}$ est définie par :
- $c_1(\mathcal{E}) = \mathcal{E} \cup \{Y \to \bullet\gamma \mid X \to \alpha \bullet Y\beta \in \mathcal{E}$ et $Y \to \gamma$ est une règle$\}$
-
- On note $c_{i+1}(\mathcal{E}) = c_1(c_i(\mathcal{E}))$: on a
  - $\mathcal{E} \subset c_1(\mathcal{E}) \subset \ldots \subset c_p(\mathcal{E}) = c_{p+1}(\mathcal{E})$
- Par définition, la clôture de $\mathcal{E}$ est $c_p(\mathcal{E})$.

État de départ

Par définition, l'état de départ de l'analyseur est la clôture de l'item S' → •S $.

Exemple :

S' → S$	S → T N
T → int	T → float
N → N [nb]	N → id

- Les clôtures successives de l'ensemble $\mathcal{E}$ = {S' → •S $} sont :
  - $c_1(\mathcal{E})$ = {S' → •S $, S → •TN}
  - $c_2(\mathcal{E})$ = {S' → •S $, S → •TN, T → •int, T → •float
  - $c_3(\mathcal{E}) = c_2(\mathcal{E})$: c'est la clôture de $\mathcal{E}$.

L'automate LR(0) : un exemple

(1) S → E$
(2) E → E+T
(3) E → T
(4) T → x
(5) T → (E)

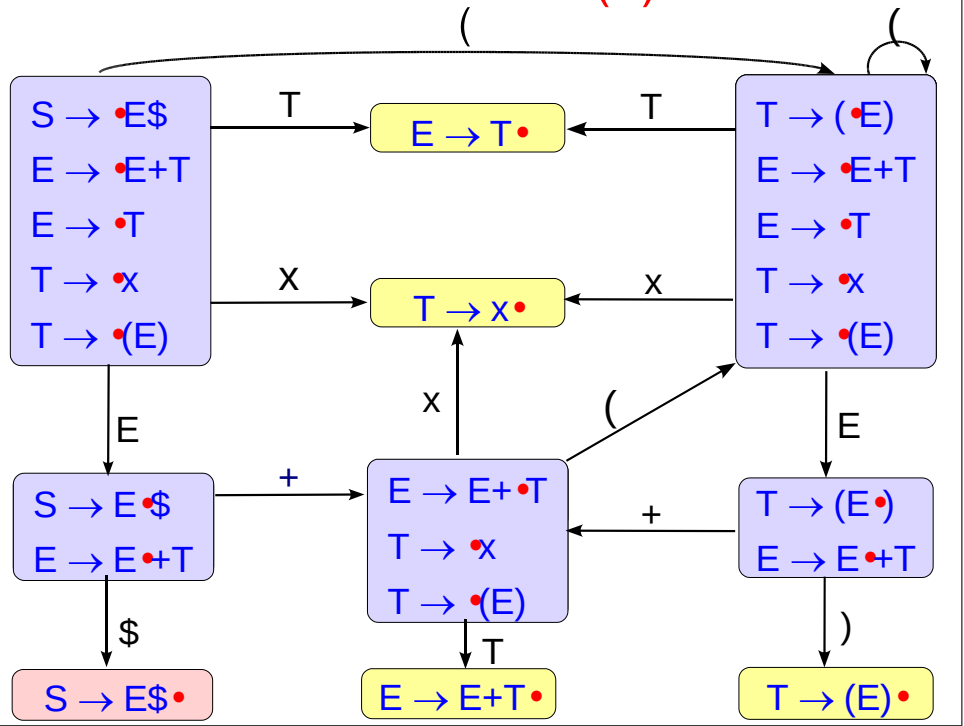
État de départ e_0 :

S → •E$
E → •E+T
E → •T
T → •x
T → •(E)

Pas de réduction possible dans l'état e_0 :

état d'empilement

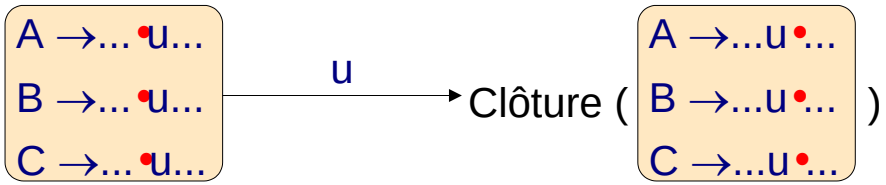
Automate LR(0)



Construction de l'automate

État initial : clôture de S' → •S$.

Transition étiquetée u (terminal ou non-terminal) :



États et action

- Rappel : l'analyseur est dans l'état empilé en haut de la pile des états.
- Si l'analyseur est dans un état
  - dans lequel il y a un seul item,
  - tel que l'item est de la forme X → α •,
  - alors il ne peut que réduire par la règle X → α.
- Si l'analyseur est dans un état
  - ne contenant pas d'item de la forme X → α •,
  - alors il ne peut qu'empiler.

Conflits LR(0)

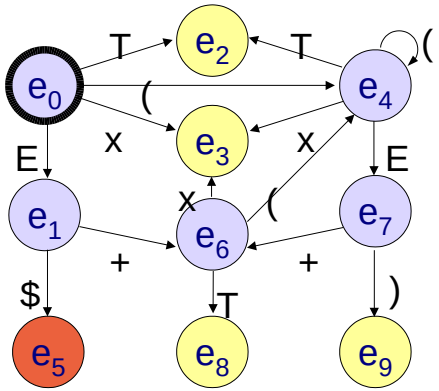
- Si l'automate a un état à plusieurs items, et
  - dont un des items est de la forme X → α •,
  - dont un autre item est de la forme Y → α' •aβ,
  - alors il y a un conflit empilement/réduction (shift/reduce)
- Si l'automate a un état à plusieurs items, et dont l'un des items est de la forme X → α •, dont un autre item est de la forme Y → β •, alors il y a un conflit réduction/réduction (reduce/reduce)

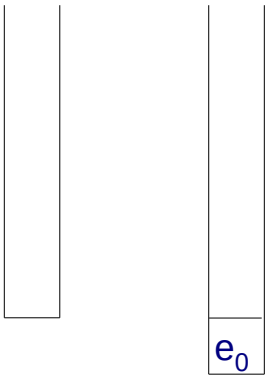
Grammaires LR(0)

Une grammaire est LR(0) si son automate LR(0) ne présente pas de conflit.

- Remarques :
  - Si une grammaire n'est pas LR(0), il peut exister une autre grammaire qui engendre le même langage et qui est LR(0).
  - Une grammaire ambiguë n'est jamais LR(0).

Utilisation de l'automate par l'analyseur

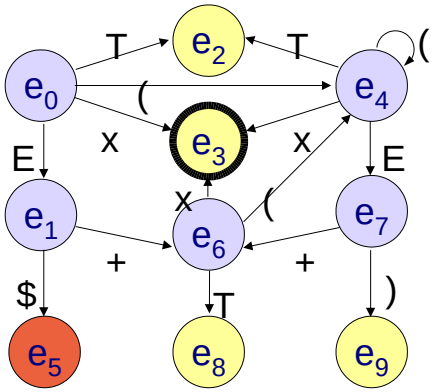


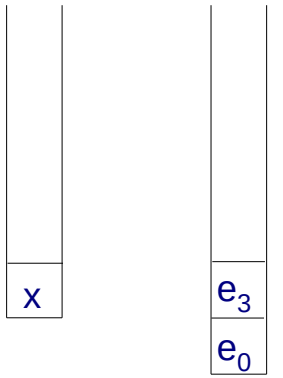


Entrée : x + (x) $

Action : empiler

Utilisation de l'automate par l'analyseur



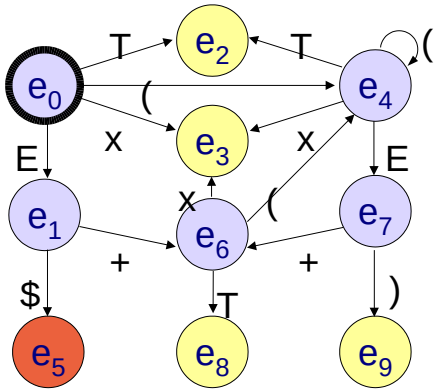


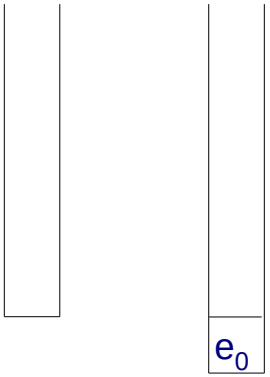
Entrée : + (x) $

Action : réduire T → x

x

Utilisation de l'automate par l'analyseur



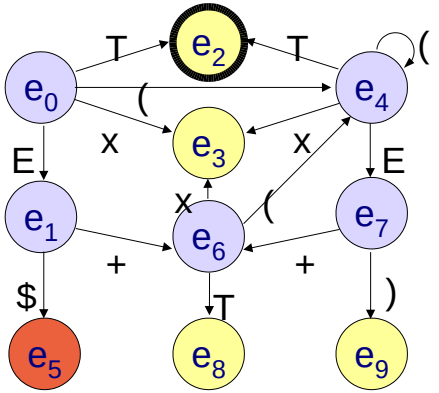


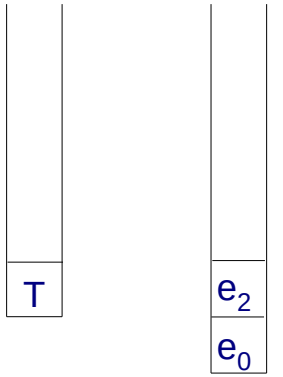
Entrée : + (x) $

En cours de réduction T → x

x

Utilisation de l'automate par l'analyseur





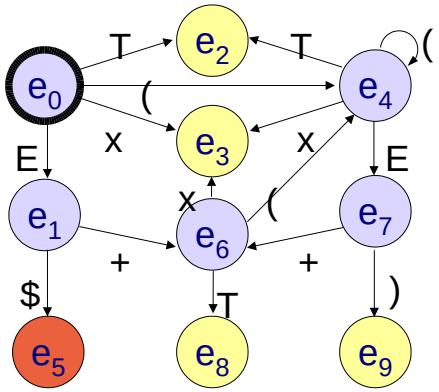
Entrée : + (x) $

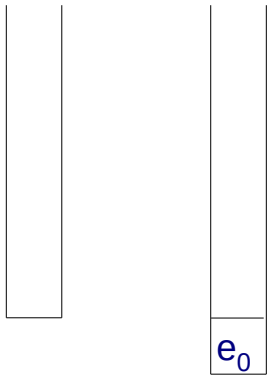
Action : réduire E → T

T

x

Utilisation de l'automate par l'analyseur

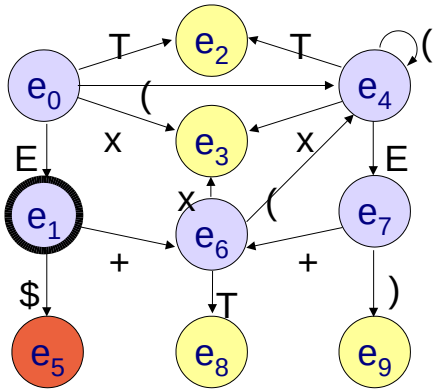


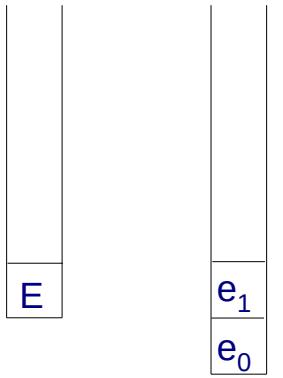


Entrée : + (x) $

En cours de réduction E → T

Utilisation de l'automate par l'analyseur

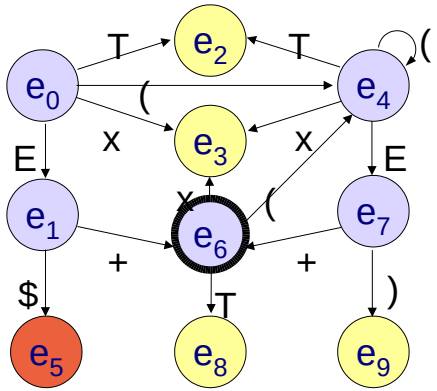


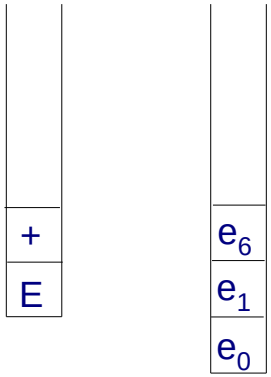


Entrée : + (x) $

Action : empiler

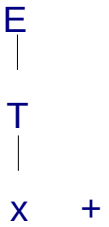
Utilisation de l'automate par l'analyseur



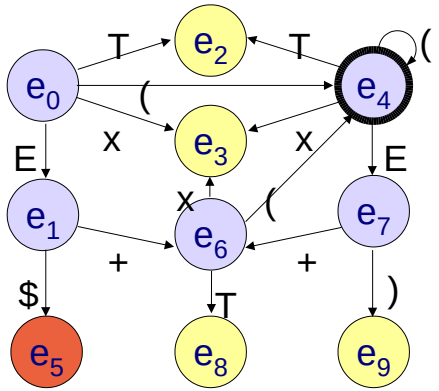


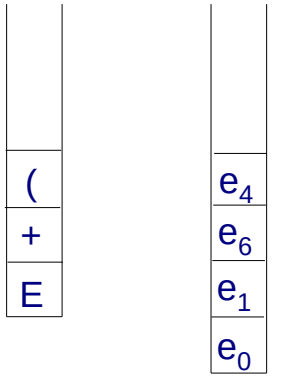
Entrée : (x) $

Action : empiler



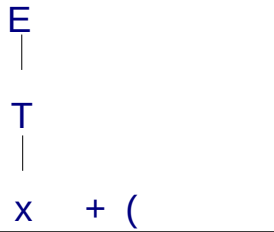
Utilisation de l'automate par l'analyseur



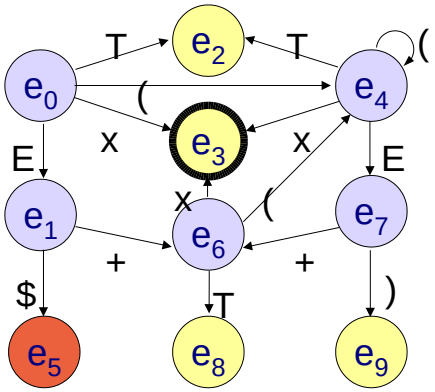


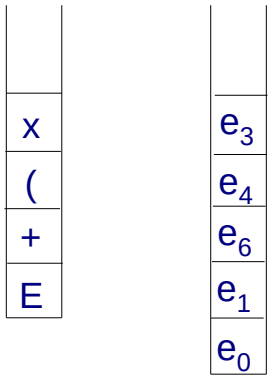
Entrée : x) $

Action : empiler



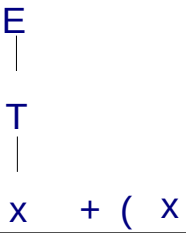
Utilisation de l'automate par l'analyseur



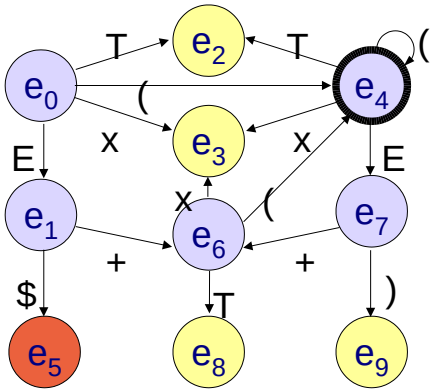


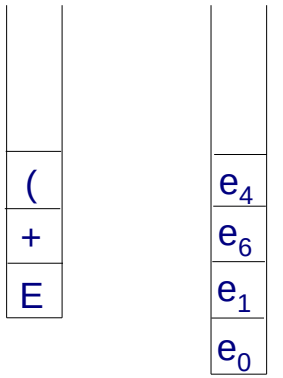
Entrée :) $

Action : réduire par T → x



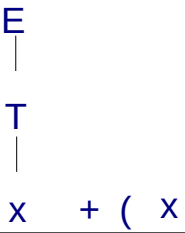
Utilisation de l'automate par l'analyseur



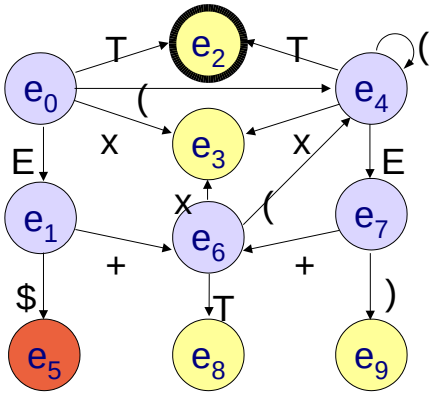


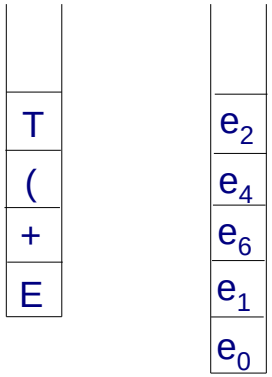
Entrée :) $

En cours de réduction T → x



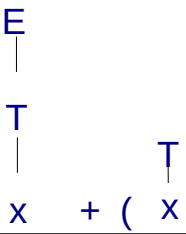
Utilisation de l'automate par l'analyseur



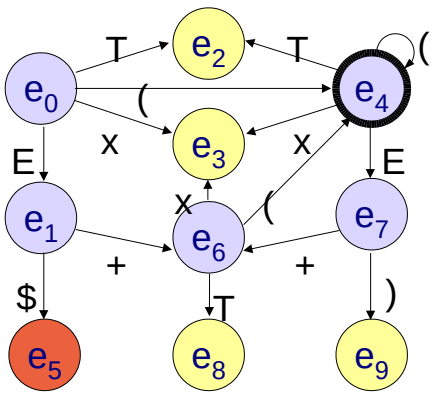


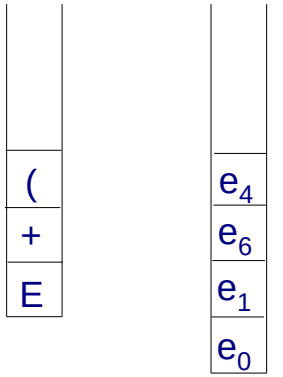
Entrée :) $

Action : réduire par E → T



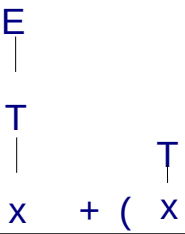
Utilisation de l'automate par l'analyseur



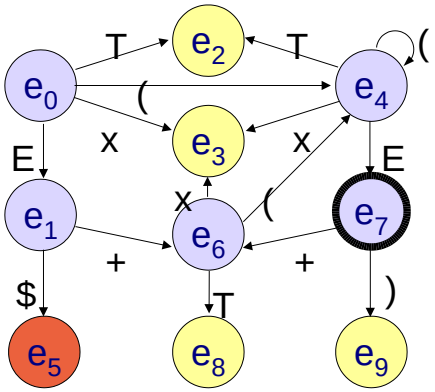


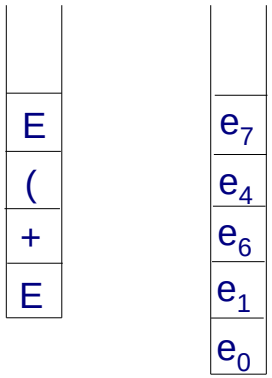
Entrée :) $

En cours de réduction E → T



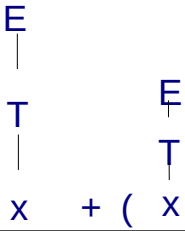
Utilisation de l'automate par l'analyseur

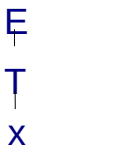




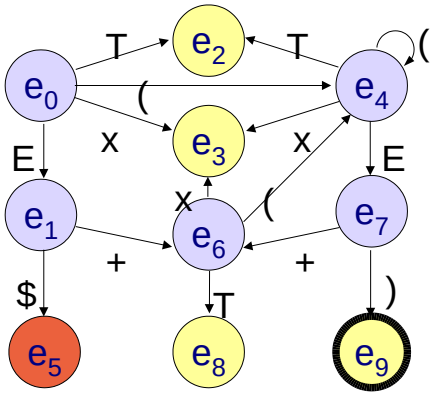
Entrée :) $

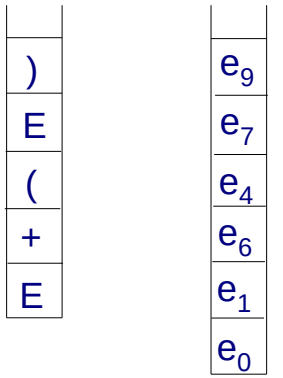
Action : empiler





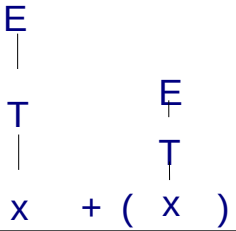
Utilisation de l'automate par l'analyseur



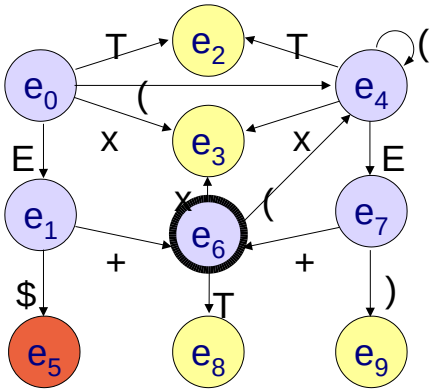


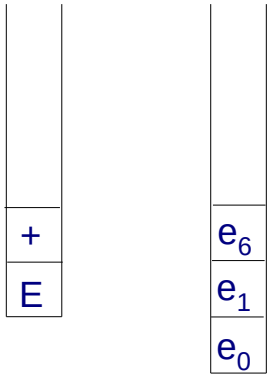
Entrée : $

Action : réduire T → (E)



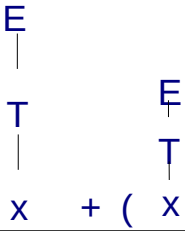
Utilisation de l'automate par l'analyseur



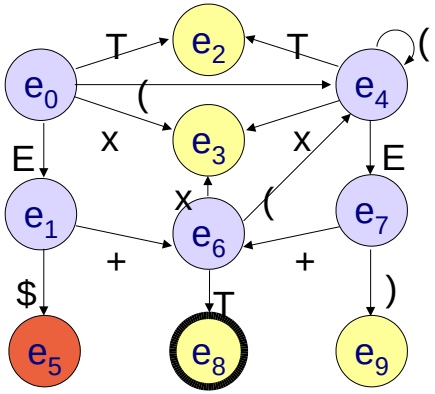


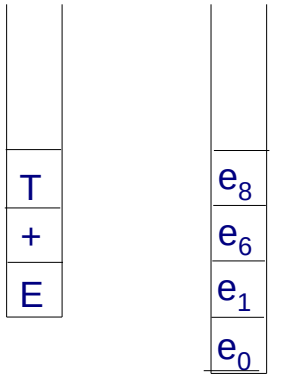
Entrée : $

En cours de réduction T → (E)



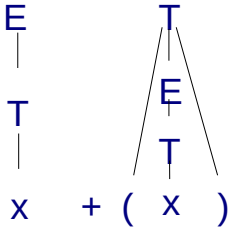
Utilisation de l'automate par l'analyseur



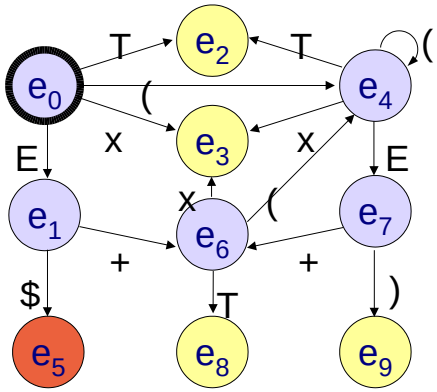


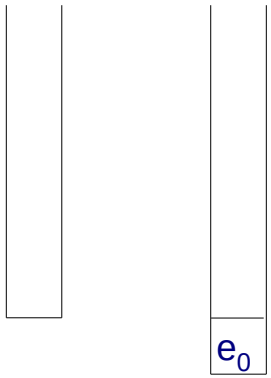
Entrée : $

Action : réduire E → E+T



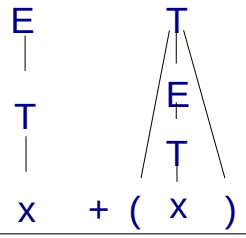
Utilisation de l'automate par l'analyseur



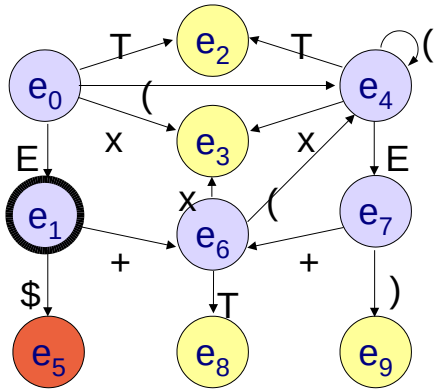


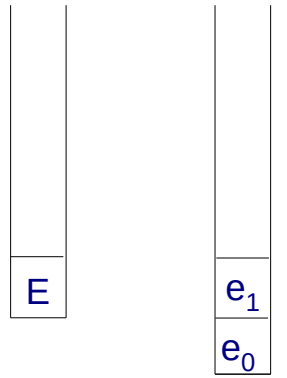
Entrée : $

En cours de réduction E → E+T



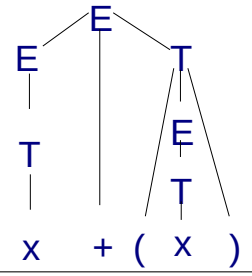
Utilisation de l'automate par l'analyseur



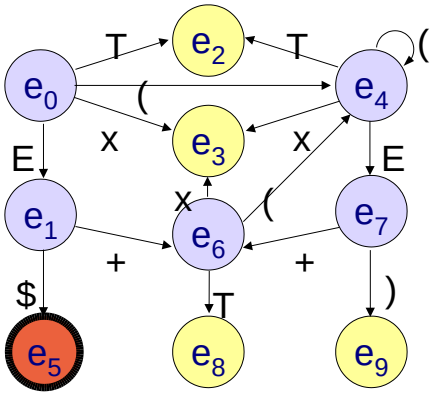


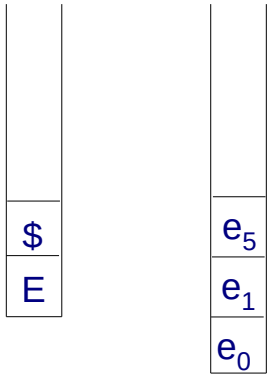
Entrée : $

Action : empiler



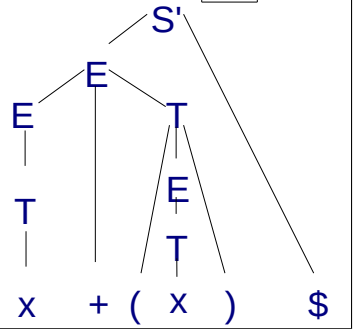
Utilisation de l'automate par l'analyseur





Entrée :

Action : accepter



Comment lever les conflits ? L'algorithme SLR(1)

- L'algorithme SLR(1) est une amélioration de l'algorithme LR(0).
- Il se sert d'un symbole d'avance.
- Il est basé sur la remarque suivante :
  Il est inutile d'essayer une réduction par une règle

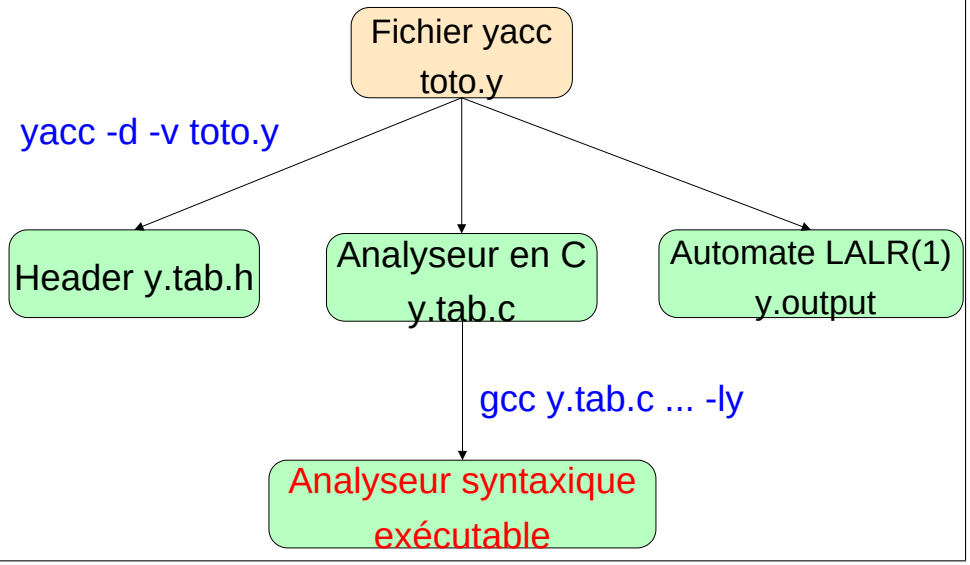
  $A \rightarrow \alpha$

  si le symbole d'avance n'est pas dans

  Suivant(A)

Yacc : Yet Another Compiler Compiler

Logiciel pour construire automatiquement un analyseur syntaxique.



Yacc : fonctionnement de base

- On décrit la grammaire en spécifiant
  - ses terminaux (ou lexèmes),
  - ses règles.

On associe à chaque règle une action qui sera exécutée lors de la réduction.

Format d'un source yacc

```
%{
Bloc littéral : déclarations C
%}
```
} Optionnel

Déclarations yacc

%%

Règles + actions

%%

} Optionnel cf. lex

Déclarations yacc : première approche

Yacc voit les terminaux comme des entiers (**`int`**)

On peut donner un nom à un terminal.

On déclare les terminaux ayant un nom :

`%token` <NOM>

Déclaration (optionnelle) du symbole de départ

`%start` <NOM>

Règles

Un ensemble de règles est donné sous la forme

<non terminal> : <membre droit$_1$>
| <membre droit$_2$>
.....
;

Exemple :

```
expr : expr '+' terme
     | terme
     ;
terme : NOMBRE
      ;
```

Interface lex ↔ yacc

- La fonction d'analyse syntaxique s'appelle **`yyparse()`**
- Elle appelle `yylex()`.
- Lorsque `yylex` reconnaît un lexème, il retourne un entier pour décrire le type de lexème rencontré
- Cet entier correspond à un terminal pour **`yyparse`**
- Les terminaux nommés via **`%token`** sont #définis dans **`y.tab.h`**, lisible par l'analyseur lexical.
- `yylex` peut aussi transmettre des informations sur la valeur du lexème via la variable **`yylval`**.

Attributs

- Chaque noeud de l'arbre porte de l'information.
- Un attribut associé à un symbole de grammaire permet de décrire comment calculer l'information.
- Exemple :
  - Grammaire des expressions.
    On peut définir un attribut valeur pour E, T, F, ID.
  - Grammaire d'un langage de programmation :
    tout symbole peut avoir un attribut code généré

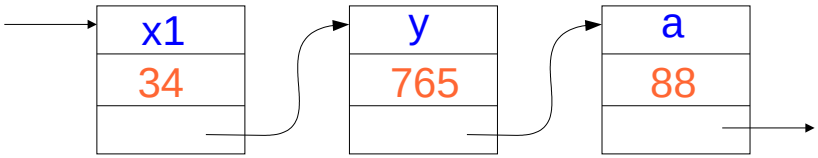
Types d'attributs

- Sous yacc, le type par défaut d'un attribut est **`int`**
- Si on veut manipuler des attributs d'un type différent, on définit les types possibles d'un attribut de la grammaire dans une directive **`%union`** :
  -
```
%union {
    <type 1> <nom de type 1>;
    <type 2> <nom de type 2>;
    ..........
}
```

Exemple :

```
%union {
    int entier;
    double reel;
}
```

Exemple : manipulation de variables




```
struct variable {
       char              *nom;
       int               valeur;
       struct variable *suivante;
     };

%union {
     int               val ;
     struct variable *var;
}
```

Types d'attributs

On peut attribuer un type à l'attribut associé à un symbole de la grammaire.

```
%token <type> nom de token
%type  <type> nom de non-terminal
```

Exemple :

```
%union {
        float reel;
        struct variable *var;
}
%token <var> IDENTIFICATEUR
%type  <reel> expression
```

Gestion des conflits

- Si la grammaire a des conflits LALR(1), on peut choisir de forcer telle réduction ou l'empilement.
- On peut ainsi traiter des grammaires ambiguës.
- Règles par défaut
  - En cas de conflit shift/reduce, l'analyseur généré par yacc choisit le shift par défaut.
  - En cas de conflit reduce/reduce, la première règle est choisie.
    Les conflits reduce/reduce sont à éviter.

Opérateurs, associativité et précédence

- Définir des précédences et associativités pour les lexèmes permet d'obtenir un comportement autre que celui choisi par défaut.
- La précédence d'un lexème est déterminée par la position de sa définition : plus il est défini tard, plus forte est sa précédence.
- Un lexème est :
  - associatif à gauche s'il a été défini par **`%left`**
  - associatif à droite s'il a été défini par **`%right`**
  - non associatif s'il a été défini par **`%nonassoc`**

Opérateurs, associativité et précédence

- La précédence d'une règle est celle de son symbole le plus à droite ayant une précédence.
- Le comportement lors d'un conflit shift/reduce n'est pas celui par défaut lorsque :
  - Le symbole d'avance x a une précédence α.
  - La règle de réduction a une précédence β.
- Si $\alpha > \beta$, l'empilement (shift) est choisi.
- Si $\alpha < \beta$, la réduction est choisie.
- Si $\alpha = \beta$,
  - si x est associatif gauche : réduction
  - si x est associatif droit : empilement
  - si x est non-associatif : erreur syntaxique

Le traitement des erreurs

Lorsque l'analyseur rencontre une erreur :

1) Il appelle yyerror ("syntax error");
2) Il dépile les symboles analysés jusqu'à ce qu'il puisse empiler le symbole **`error`**
3) Il consomme le symbole qui a provoqué l'erreur
4) S'il rencontre une autre erreur avant d'avoir pu empiler 3 symboles, il revient en 2).

- La fonction yyerror peut être redéfinie.
- On peut forcer la resynchronisation avec yyerrok; L'analyseur considère alors qu'il n'est plus en mode de reprise d'erreur.
- YYRECOVERING() indique si l'analyseur est en reprise d'erreur.

Attributs

- Attribut : information attachée à un noeud de l'arbre syntaxique.
- Exemples :
  - valeur entière (expression, variable),
  - nom : chaîne de caractères (variable),
  - adresse en mémoire (fonction, variable),
  - code généré (suite d'instructions),
  - type (variable).
- Un symbole (terminal ou non) peut avoir plusieurs attributs différents.

Traduction dirigée par la syntaxe

- Une grammaire attribuée est une grammaire dans laquelle :
  - chaque symbole, terminal ou non, peut avoir des attributs,
  - chaque attribut possède des règles de calcul en fonction d'autres attributs et de valeurs initiales,
- chaque règle de calcul est attachée à une règle de la grammaire.

$$A \rightarrow \alpha \qquad a = f(a_1,...,a_n)$$

La fonction f, sans effet de bord, calcule a.

- But : calculer les attributs pendant l'analyse syntaxique

Attributs synthétisés

- Soit la règle calculant l'attribut a associée à $A \rightarrow \alpha$

  $A \rightarrow \alpha \qquad a = f(a_1,...,a_n)$

- a est synthétisé si c'est un attribut de A calculé en fonction des attributs des symboles de α.

- Exemple : E.val est synthétisé.

- $E \rightarrow E' + E''$ $\qquad$ E.val = E'.val + E''.val
  $E \rightarrow E' * E''$ $\qquad$ E.val = E'.val * E''.val
  $E \rightarrow num$ $\qquad$ E.val = num.val.

- L'information « remonte » dans l'arbre syntaxique

Attributs hérités

- Soit la règle calculant l'attribut a associée à $A \rightarrow \alpha$

  $A \rightarrow \alpha \qquad a = f(a_1,...,a_n)$

- L'attribut a est hérité
  - si c'est un attribut d'un symbole de α, et
  - $a_1,...,a_n$ sont des attributs de symboles de α ou de A.

Grammaires S-attribuées

- Une grammaire est S-attribuée si tous ses attributs sont synthétisés.
- Grammaire S-attribuée $\Rightarrow$ calcul des attributs de façon mécanique si on a un analyseur LR.
- Lors de la réduction par

  $A \rightarrow \alpha$

  on calcule tous les attributs en A en fonction des attributs aux symboles de α.
- On peut utiliser une pile pour calculer les attributs

Calcul des attributs synthétisés

Utilisation d'une pile.

Exemple :

E	E.val = 1
+	
E	E.val = 2

Après réduction par $E \rightarrow E+E$

E	E.val = 3

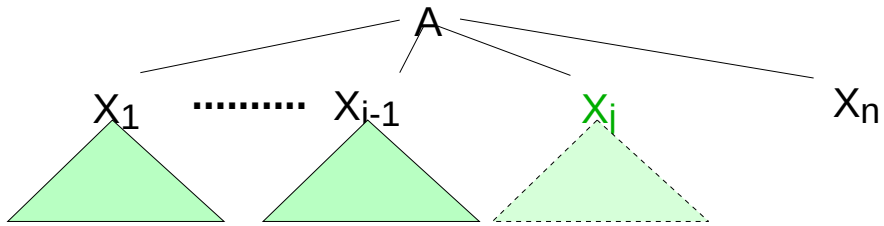
Grammaires L-attribuées

- Une grammaire est L-attribuée si tout attribut hérité de X_j calculé par une règle associée à

$$A \rightarrow X_1 \cdots X_n$$

dépend seulement
  - des attributs de $X_1,...,X_{j-1}$,
  - des attributs hérités de A
- Grammaire L-attribuée $\Rightarrow$ calcul des attributs de façon mécanique si on a un analyseur LL.
- Une grammaire S-attribuée est L-attribuée

Grammaires L-attribuées



- Lorsque l'on doit prédire la règle $X_j \rightarrow \alpha$,
  - les attributs de $X_1,...,X_{j-1}$ sont connus,
  - les attributs hérités de A sont connus.

A() pour j = 1 à n
  calculer les attributs hérités de X_j
  appeler $X_j()$;
calculer les attributs synthétisés de A

Table des symboles

Pendant l'analyse, on utilise une table mémorisant les informations relatives aux symboles.

Exemple : Pseudo-pascal, fonctions imbriquées

```
x1: int;
function f(a: int): int
{
  function g(b: float): char
  {
     x: static int;
     .....
  }
  y : char;
  .....
}
```

x1	Variable	glob	int	0
f	Fonction		int	

a	Paramètre		int	-1
y	Variable	dyn	char	0
g	Fonction		char	

b	Paramètre		float	-4
x	Variable	stat	int	1

Attributs hérités et yacc

Yacc permet de calculer des attributs synthétisés

$$\$\$ = f (\$1,...,\$n)$$

On peut gérer sous yacc des attributs hérités, en accédant à des attributs empilés plus bas que le membre droit de la règle de réduction :

α_n	$n
...	...
α_1	$1
	$0
	$-1

Pile avant réduction par

$$A \rightarrow \alpha_1 \cdots \alpha_n$$

Attributs hérités et yacc

Exemple :

(1)D → T S ;	S.type = T.type
(2)S → id	id.type = S.type
(3)S → S' , id	id.type = S.type
(4)T → int	T.type = int
(5)T → float	T.type = float

int creersym(char *nom, int type);

(1)D → T S ;	creersym($1, $0)
(2)S → id	creersym($3, $0)
(3)S → S' , id	
(4)T → int	$$ = INT
(5)T → float	$$ = FLOAT

Appel de fonction

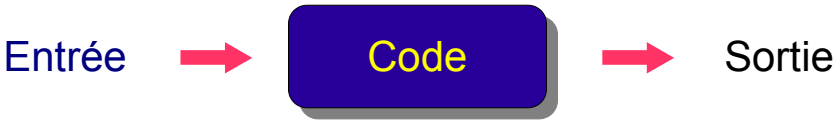
- A l'appel, la fonction appelante :
  - laisse de la place sur la pile pour la valeur retour de la fonction appelée,
  - empile le registre de base courant, le CP,
  - empile les paramètres de la fonction appelée,
  - met à jour le registre de base,
  - passe le contrôle à la fonction appelée qui empile ses arguments.
- Au retour, la fonction appelée
  - écrit sa valeur retournée dans la place prévue,
  - dépile ses arguments,
  - restaure les registres.

Interpréteur

- Principe d'un interpréteur :

- Avantage : pas de précalcul nécessaire sur le programme.
- Inconvénient : une même partie du programme peut être analysée plusieurs fois pendant l'exécution.

Les traducteurs



- Deux phases :
  - Compilation du programme afin de produire un programme machine.
  - Exécution du programme machine sur l'entrée.

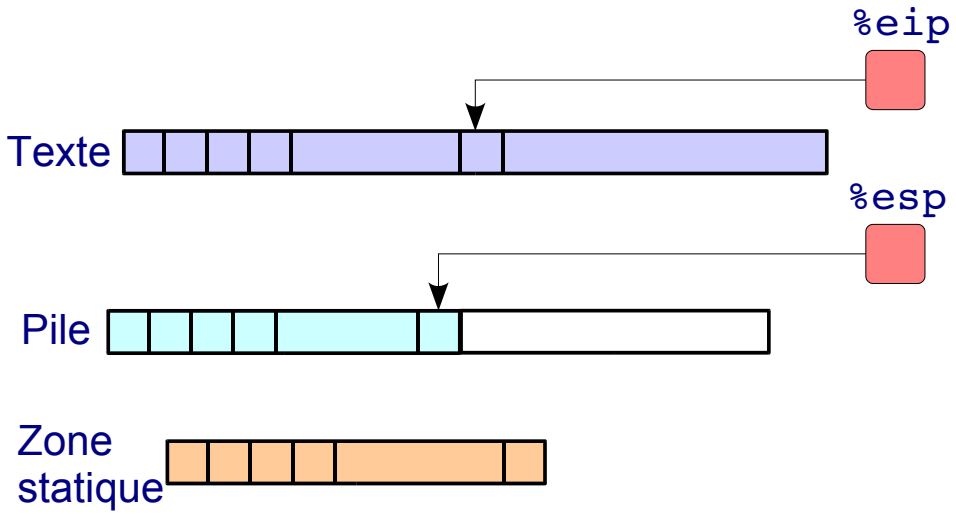
Traduction de code

- Inconvénients de la traduction :
  - La traduction prend elle-même du temps.
- Avantages :
  - La traduction ne se fait qu'une fois par programme
  - L'exécution du programme est plus efficace, ce qui augmente l'intérêt pour des programmes à lancer plusieurs fois.

Les langages intermédiaires

- Dans une première phase de traduction, on traduit vers un langage intermédiaire.
- Avantages : modularité, factorisation de la traduction
- Une machine virtuelle représente une abstraction idéalisée du matériel. Elle peut posséder :
  - des registres,
  - une pile,
  - une zone statique,
  - une zone de texte.
- La machine virtuelle exécute les instructions écrites en langage intermédiaire.

Une C-machine

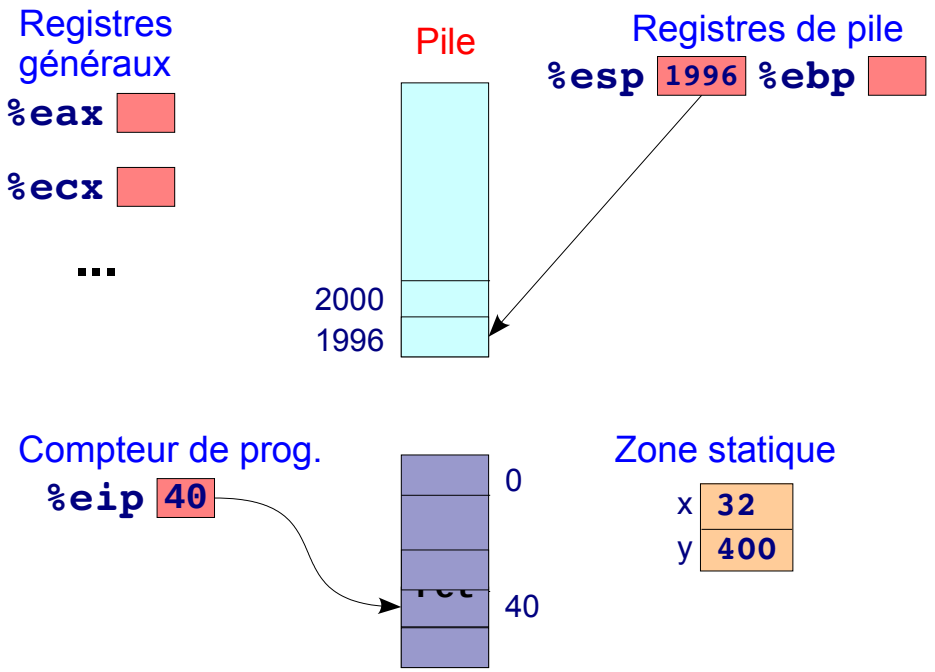


Registres : `%eax` : accumulateur
`%esp` : sommet de pile
`%eip` : pointe sur l'instruction suivante

Machine virtuelle, langage intermédiaire

- Toutes les données manipulées par la machine virtuelle sont entières et de même taille.
- Pour tester le code généré sans écrire un interpréteur, le langage choisi est un assembleur réel
- Détails techniques de l'assembleur :
  - La taille de toute donnée sera ici toujours 4.
  - La pile sera représentée « à l'envers ».
  - Adresses les plus grandes en haut de pile.
  - La zone statique est définie dans le programme
  - On peut déclarer dans cette zone un nombre fini de symboles globaux.

Machine virtuelle



Fonctionnement de la machine

- La machine charge l'instruction `Text[%eip]` dans un registre d'instruction **IR**, et l'exécute.
- Avant l'exécution, le registre **%eip** est incrémenté

```
while(true)
    {
        IR = Text[%eip];
        %eip++;
        executer(IR);
    }
```

- L'utilisateur ne manipule pas `%eip` directement, mais via instructions spéciales (**call/ret**, sauts)

Le langage intermédiaire

- Les instructions du langage choisi ont 0, 1 ou 2 opérandes.
- C'est une sous-partie d'un assembleur réel (80x86, syntaxe AT&T).
- Note : gcc peut compiler vers notre langage intermédiaire.
  - Exemple, sur garbanzo :

```
$ export PATH={PATH}:/usr/local/lib\
/gcc-lib/i486-elf-sysv4/2.7.0

$ gcc -S -b i486-elf-sysv4 titi.c
```

Modes d'adressage (r-valeurs)

- Le langage intermédiaire dispose de plusieurs façons pour spécifier une r-valeur :

Immédiat	**$7**	Constante 7
Direct	**%eax**	Contenu de %eax
Indirect	**(%eax)**	Contenu de l'emplacement d'adresse **%eax**
Indexé	**4(%eax)**	Contenu de l'emplacement d'adresse **%eax+4**

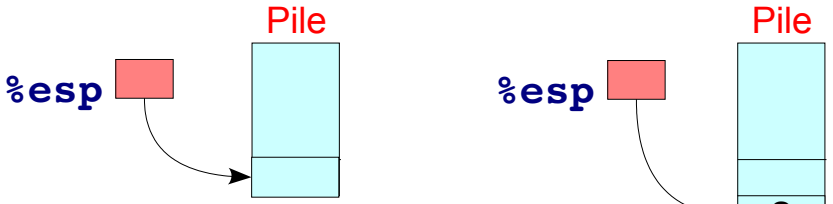
Modes d'adressage (l-valeurs)

- Le langage intermédiaire dispose de plusieurs façons pour spécifier une l-valeur :

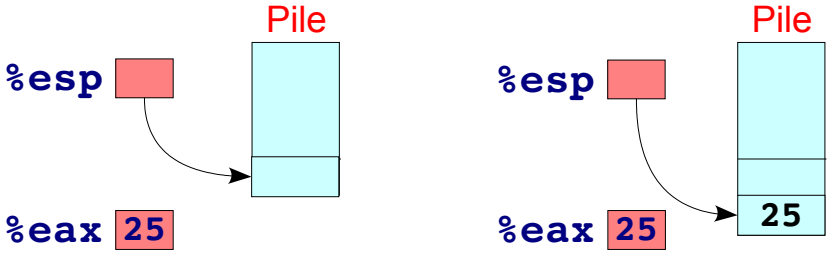
Nom du mode	Notation	Valeur
Immédiat	Interdit !	
Direct	`%eax`	`%eax`
Indirect	`(%eax)`	Emplacement d'adresse `%eax`
Indexé	`-4(%eax)`	Emplacement d'adresse `%eax-4`

Instructions (1)

- `pushl x` : empiler `x` (r-valeur)
- Exemple : **`pushl $3`**

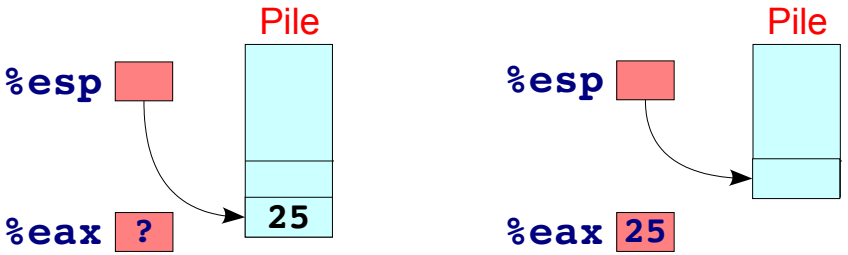


- Exemple : **`pushl %eax`**



Instructions (1)

- `popl x` : dépiler dans `x` (l-valeur)
- Exemple : `popl $3` : interdit !
- Exemple : `popl %eax`



Instructions arithmétiques

- **`movl x,y`** déplace (le contenu de) `x` dans `y`
  `c'est-à-dire : y = x`
- **`addl x,y`** `y = x + y;`
- **`subl x,y`** `y = x - y;`
- **`negl x`** `x = -x;`
- **`incl x`** `x++;`
- **`decl x`** `x--;`
- **`imull x,y`** `y = x * y`
- **`idivl x,y`** `y = x / y`
- Les instructions interprètent x et y comme des entiers signés.

Instructions de saut

- Une instruction peut être étiquetée.
- **<nom_etiquette>:** dans source étiquette l'instruction suivante par **<nom_etiquette>**.
- **cmpl x,y** compare **y** à **x** en vue d'un test ultérieur.
- Une instruction de saut change la valeur de **%eip**, si `y - x` vérifie une certaine condition.
- La nouvelle valeur de **%eip** est donnée dans l'instruvtion de saut.

Instructions de saut

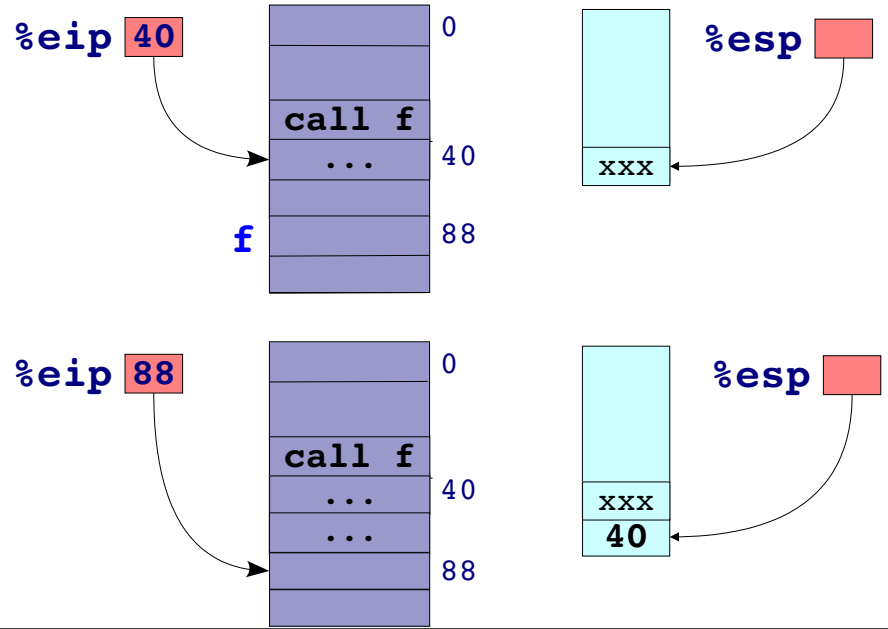
jmp .l	**saut à l'instruction d'étiquette .l**
je .l	**idem, si y = x**
jne .l	**idem, si y ≠ x**
jl .l	**idem, si y < x**
jle .l	**idem, si y ⩽ x**
jg .l	**idem, si y > x**
jge .l	**idem, si y ⩾ x**

Sous-programmes

- Le langage a deux instructions spéciales pour implémenter simplement les appels de fonctions.

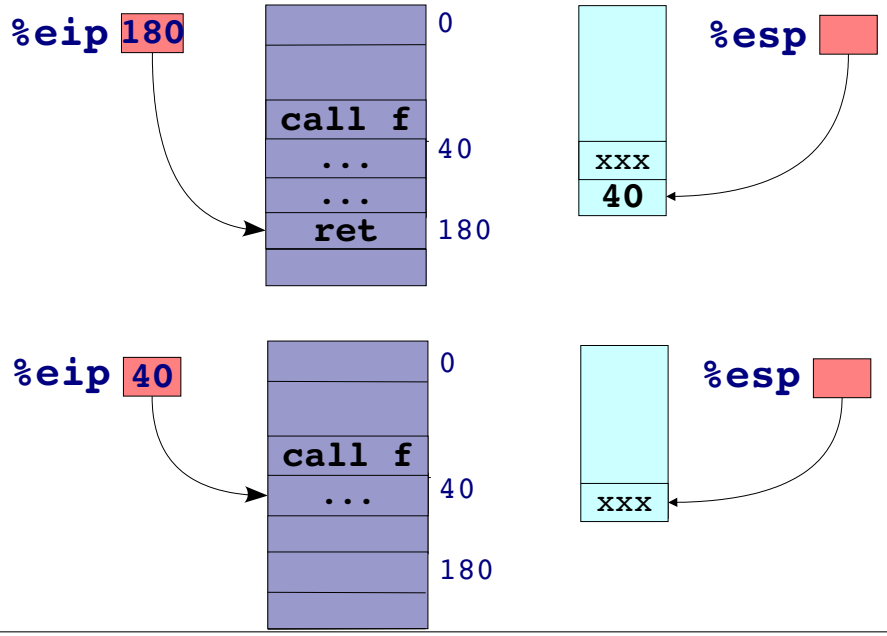
L'instruction **call**

- **call f** empile la valeur courante de **%eip** et saute à l'instruction étiquetée par l'étiquette **f**.



L'instruction `ret`

- **`ret`** dépile dans `%eip` : équivaut à **`popl %eip`**



Transmission des arguments

- Supposons que la fonction **`f`** appelle `g(a,b)` :
- À l'intérieur du code de **`f`**

```
pushl b              # empiler b
pushl a              # empiler a
call  g              # appeler g
addl  $8,%esp        # dépiler a et b
```

- C'est la fonction appelante qui empile les arguments, à l'envers.
- Ceci permet de traiter les fonctions à nombre variable d'arguments, car le premier argument est à distance fixe de `%ebp`.

Accès aux arguments

- Comment accéder aux arguments dans une fonction ?
- Une première possibilité serait d'indexer leur position par rapport au registre de pile.
- Inconvénient : ce registre n'est pas fixe durant tout le déroulement de la fonction.
- On indexe par rapport à un registre que l'on maintient fixe durant tout le déroulement de la fonction, le registre de base (%ebp dans les exemples).
- Les variables locales dynamiques s'indexent également par rapport à ce registre.