Introduction
à la
compilation
(I)

Plan

1. Généralité
2. Analyse lexicale
   - Automate fini* (déterministe / non déterministe)
   - Expression rationnelle (régulière, regular expression)
   - Lex
3. Analyse syntaxique : LL(1), …, LALR(1)

   Yacc
4. Présentation du MiniLangage, du processeur MIPS et de l'environnement SPIM
5. Génération de code : structures de contrôle

** juste énoncé, mais ne fait pas partie de ce cours.*

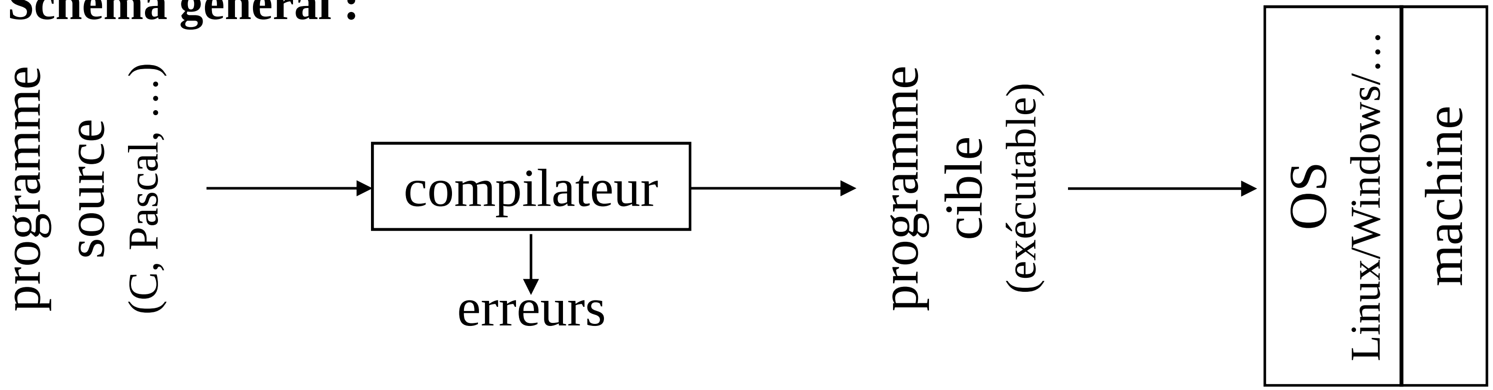
6. Génération de code : environnement global et structures de données
7. Génération de code : environnement local (blocs)
8. Génération de code : appel de procédures et fonctions
9. Génération de code : passage des paramètres
10. Machine abstraite
11. Implantation d’une machine abstraite

1. Généralité

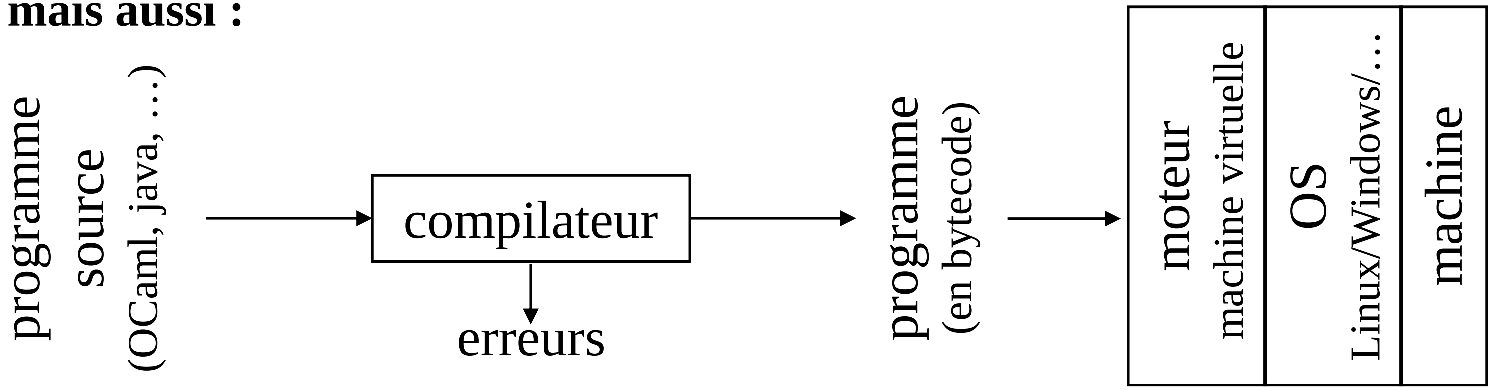
- **Définition : (simplifiée, mais générale)**

  Un compilateur est un programme exécutable qui lit un programme écrit dans un langage (source) et le traduit en un programme écrit dans un autre langage (cible)

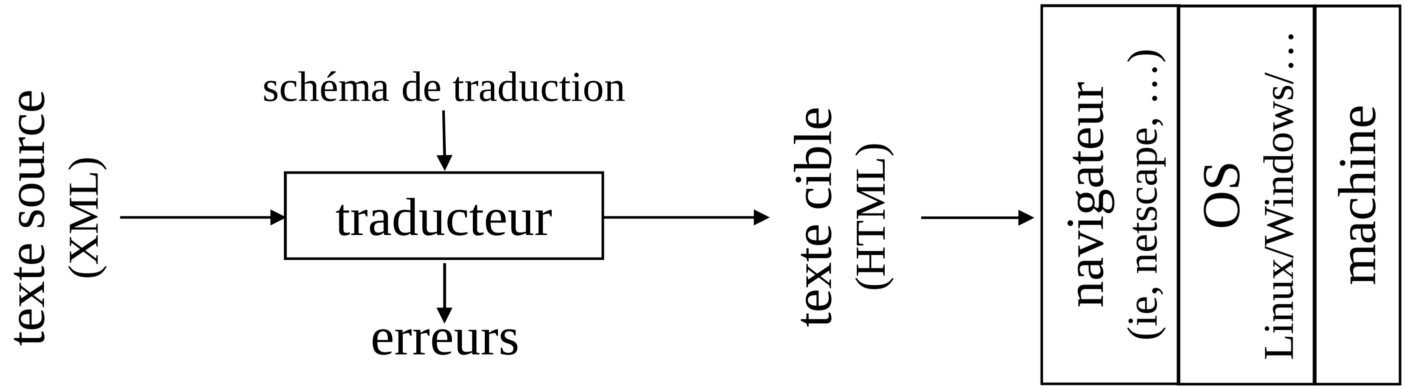
- **Schéma général :**



- **mais aussi :**

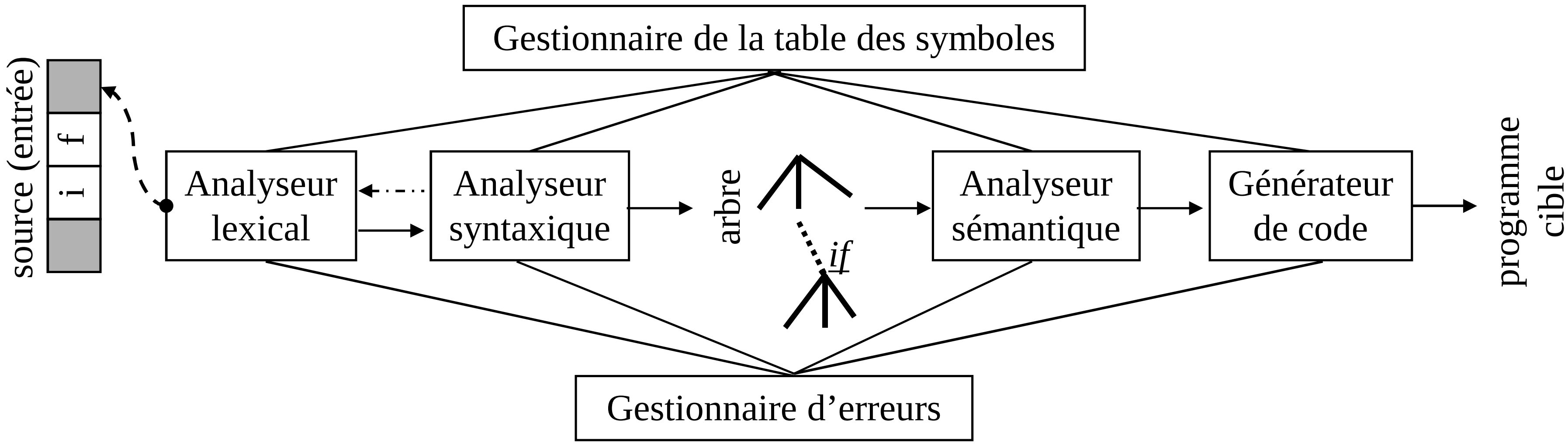


- **ou encore : (pas trop différent)**



- **et bien d'autres schémas encore, …**

- **Compilateur :**



- **Dans ce cours : (Cours de 1 à 9)**

Programme écrite en
MiniLangage
(avec des variables, structures de contrôle, fonctions et procédures)

Compilateur

Langage Source → *Langage Cible*

MiniLangage → **MIPS**

écrit en

C

(Simulateur de MIPS)

spim & xspim (1e)

écrit en **C**

Programme Source écrit en MiniLangage

```
VAR n, r;

ECRIRE "Donner un entier :\n";

LIRE n;
r := 1;

ECRIRE "La valeur de la
factorielle de ";
ECRIRE n;
ECRIRE " est ";

TANTQUE 0 < n FAIRE {
  r := r * n;
  n := n - 1
};

ECRIRE r;
ECRIRE ".\n"
.
```

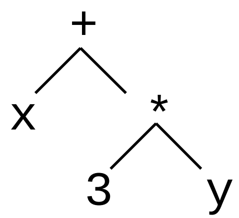
Compilateur

Programme Cible généré en MIPS

```
        .data
MEM: .space 8
CHAINE0: .asciiz "Do..... :\n"
      .....
        .text
main:    la $30, MEM
        la $4, CHAINE0
        li $2, 4
      .....
        j E100
E101:   lw $8, 4($30)
      .....
E100:   li $8, 0
        lw $9, 0($30)
        slt $8, $8, $9
        bne $8, $0, E101
      .....
        li $2, 10
        syscall
```

- **Le texte source (entrée)**
  - **est linéaire, composé de caractères (saut de ligne compris) qui se suivent,**
  - **représente un programme contenant des structures de contrôles (boucle, conditionnel, séquences, etc., …) qui peuvent être vues comme des arbres.**
- **La première étape est la construction de l'arbre (arbre syntaxique abstrait dit aussi arbre abstrait). Par exemple, pour l'expression arithmétique**

x + 3 * y ➔



- **Séparation entre l'analyseur lexical et l'analyseur syntaxique :**
  - **séparer les genres :**
    - **analyse lexical :**
      - ✓ **analyse la composition des mots de l'alphabet (structure linéaire),**
      - ✓ **…**
    - **analyse syntaxique :**
      - ✓ **analyse la composition du programme (structure arborescent),**
      - ✓ **…**
  - **meilleur portabilité, plus facile à améliorer et à corriger.**

2. Analyse lexicale

- **Automate fini* (déterministe AFD / non déterministe AFN)**

  - **Définition (AFD) : quintuplet (S, Σ, T, s, A)**
    - **Σ un alphabet,**
    - **S un ensemble d'états,**
    - **$T : S \times \Sigma \rightarrow S$ une fonction de transition,**
    - **s un état de départ (initial),**
    - **A un ensemble d'états d'acceptation (finaux), avec $A \subseteq S$.**

  - **Définition (AFN) : quintuplet (S, Σ, T, s, A)**
    - **…,**
    - **$T : S \times (\Sigma \cup \{\varepsilon\}) \rightarrow P(S)$ une fonction de transition où $P(S)$ est l'ensemble des parties de S et ε est le mot vide.**
    - **…**

* *juste énoncé, ne fait pas partie de cet enseignement.*

- **<u>Exemple</u> : AFD pour le langage des entiers relatifs**
  - $\Sigma = \{ -, 0, 1, 2, \ldots, 9 \}$
  - $S = \{ S_0 , S_1 , S_2 \}$
  - $T : S \times \Sigma \rightarrow S$

    $T(S_0 , -) = S_1$

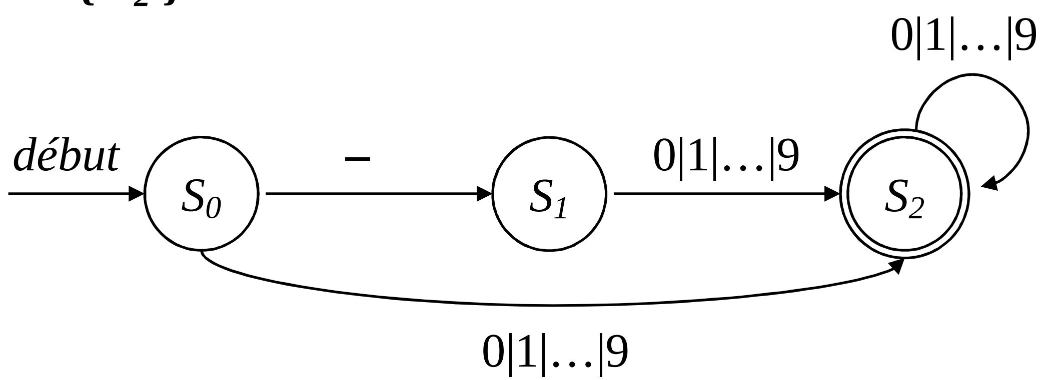
    $T(S_0 , 0) = T(S_0 , 1) = \ldots = T(S_0 , 9) = S_2$

    $T(S_1 , 0) = T(S_1 , 1) = \ldots = T(S_1 , 9) = S_2$

    $T(S_2 , 0) = T(S_2 , 1) = \ldots = T(S_2 , 9) = S_2$
  - $s = S_0$
  - $A = \{ S_2 \}$

T	**–**	0	1	…	9
S_0	S_1	S_2	S_2	…	S_2
S_1		S_2	S_2	…	S_2
S_2		S_2	S_2	…	S_2

<u>Table de transition</u>



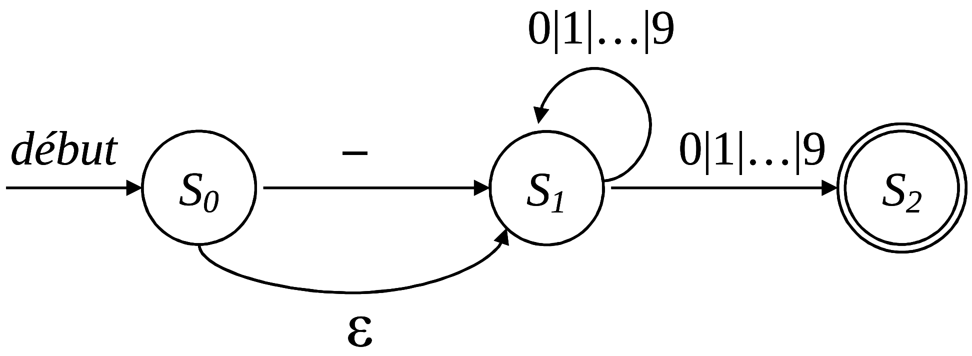
<u>Figure 2-1</u>

- **Pour un langage donné, il n’est pas facile pour un utilisateur de trouver un AFD (table de transition). En plus la table est chargée (avec beaucoup d’états).**

- **<u>Exemple</u> : AFN des entiers relatifs**
  - **Σ = { -, 0, 1, 2, …, 9 }**
  - $S = \{ S_0 , S_1 , S_2 \}$
  - $T : S \times (\Sigma \cup \{\varepsilon\}) \rightarrow P(S)$

    $T(S_0 , -) = T(S_0 , \varepsilon) = \{ S_1 \}$

    $T(S_1 , 0) = T(S_1 , 1) = \ldots = T(S_1 , 9) = \{ S_1 , S_2 \}$
  - $s = S_0$
  - $A = \{ S_2 \}$



<u>*Figure 2-2*</u>

- **Un peu plus facile à trouver qu'un AFD et en général avec moins d'états.**

- **<u>Théorème de Kleene</u>* (résultat) : les langages reconnus par les automates finis (déterministes ou non) sont exactement les langages qui peuvent être décrits par les expressions rationnelles.**

- **Exemple d'expression rationnelle (étendue pour sed, grep, Lex,…) pour les entiers relatifs**

	`–?[0-9]+`	(voir Figure 2-2)
ou	**`–?(0|1|2|3|4|5|6|7|8|9)+`**	
ou	**`(-[0-9]+)|([0-9]+)`**	(voir Figure 2-1)

Plus facile à écrire par un utilisateur.

- **Il existe des méthodes pour transformer ***

expression rationnelle ⇔ AFN ⇔ AFD ⇔ AFD minimal (d'états)

(voir [1] pour le côté pratique, voir [2] pour le côté un peu plus formel)

- **Le but est donc d'exprimer notre langage avec des expressions rationnelles (étendues) sous Lex : elles sont beaucoup plus faciles à utiliser dans l'écriture d'un analyseur lexical.**

- **Expression rationnelle (étendue) : (définition)**

est une expression rationnelle ER (étendue) pour Lex, Flex, grep, sed, …
(″regular expression″ en anglais, d'où le mot ″régulière″) :

- **un caractère quelconque autre que les caractères dits ″opérateurs″ qui sont** `" \ [ ] ^ - ? . * + | ( ) $ / { } % < >`

  Par exemple : `a` *pour représenter le mot* `a`.

- `\x` **Pour x =** `a` **(resp.** `b`, `f`, `n`, `r`, `t`, `v`**), il représente, comme en ANSI_C (voir [3e]), le caractère** `audible alert` **(resp.** `back space`, `form feed`, `new line` *(saut-fin de ligne)*, `carriage return`, `horizontal tabulation`, `vertical tabulation`**).**

  Pour les autres caractères, y compris les caractères opérateurs, `\` **permet de les rendre littéraux.**

  ou `\000`, `\001`, **...,** `\177` **les caractères ASCII en octal.**

  Par exemple : `\c` *représente le caractère* `c`

  `a\+` *représente le mot* `a+` *et non* *`a|aa|aaa|`…*

  `\\` *représente le caractère* `\`

  `\101` *représente le caractère* `A`.

- **`.`** **Un point.**
  Il représente n'importe quel caractère sauf le caractère correspondant au saut de ligne `newline`.

- **`"xyz"`** **représente le mot xyz composé de caractères quelconques x, y et z (même les caractères opérateurs sauf `"` et `\` qui garde leur propriété d'opérateur).**

  Par exemple : `"+\141\\\"\?? 123"` *pour représenter le mot* `+a\"?? 123` *où sont compris les caractères* `+`, `\`, `"`, `?`, *l'espace entre* `?` *et* `1` *avec* `\141` *le code octal du caractère* `a`.

- **βγ** **(produit ou concaténation) avec β et γ deux ER.**
  Pour créer une nouvelle ER en juxtaposant des ER.

  Par exemple : `abc` *et* `"\[de"` *sont des ER, alors* `abc"\[de"` *l'est.*
  Ce dernier représente le mot `abc[de`

- **β|γ** **(union ou choix) avec β et γ deux ER.**
  Pour avoir le choix entre les ER β ou γ.

  Par exemple : `abc` *et* `"\[de"` *sont des ER, alors* `abc|"\[de"` *l'est.*
  Ce dernier représente soit le mot `abc` *ou le mot* `[de`

- **β^* (étoile) avec β un ER.**

  On pourrait écrire à la place $\varepsilon \mid \beta \mid \beta\beta \mid \beta\beta\beta \mid \beta\beta\beta\beta \mid \ldots$

Par exemple : `abc` *est un ER, alors* `(abc)`* *l'est.*
Ce dernier représente

- *soit le mot vide*
- *soit* `abc`
- *soit* `abcabc`
- *soit* `abcabcabc`
- *soit …*

C'est à dire des mots composés de 0, une, deux ou autant de fois que l'on veut du mot `abc`*.*

- **β^+ avec β un ER.**

  On pourrait écrire à la place $\beta|\beta\beta|\beta\beta\beta|\beta\beta\beta\beta \mid \ldots$

  Même chose que β^*, mais en enlevant le mot vide ε.

- `[azh]` **Les ensembles de caractères (<u>Attention</u> : uniquement les caractères). C'est équivalent à** `a|z|h`**.**

- `[a-\144e5\067-9]` **Avec des intervalles.**
  C'est équivalent à `a|b|c|d|e|5|7|8|9`**.**
  (codes octaux : `\144` = `d` **et** `\067` = `7`**)**

- `[^a-e\n5-9]` **(négation) <u>tous les caractères</u> sauf les** `a`**, …,** `e`**,** `5`**, …,** `9` **et** `\n`**.**
- `(`**βγδ**`)` **pour pouvoir regrouper les ER β, γ et δ.**

  Par exemple : `0(abc)?` *représente le mot* `0` *ou* `0abc`
  `0abc?` *représente le mot* `0ab` *ou* `0abc`

- **β**`?` **(option) 0 ou 1 fois l'ER β.**

  Par exemple : `(abc)?` *représente le mot vide ou* `abc`

- **β**`{3}` **exactement 3 fois l'ER β, c'est-à-dire βββ**
- **β**`{2,}` `2` **fois l'ER β au moins, c'est-à-dire ββ|βββ|ββββ|…**
- **β**`{2,4}` **équivalent à ββ|βββ|ββββ**

- `^`**β** **pour représenter le mot reconnu par l'ER β si le mot est en début de ligne.**

  Par exemple : `^(abc)` *permet de reconnaître le mot* `abc` *si ce dernier est en début de ligne.*

- **β**`$` **la même chose que précédemment, mais en fin de ligne.**

- `<`*cond*`>`**β** **pour représenter le mot reconnu par l'ER β si l'analyse lexicale en cours se fait sous la condition** *cond*.

  On dit que β est qualifié (marqué) par la condition *cond*.

  Les conditions permettent d'activer ou de désactiver (donc choisir) les règles d'analyse à tout instant.

- `<`*$cond_1$*`,` …`,` *$cond_n$*`>`**β** **si l'analyse lexicale en cours se fait sous une ou des conditions parmi** *$cond_1$*, *$cond_i$* **ou** *$cond_n$*.
- `<*>`**β** **sous toutes les conditions (inclusives, exclusives et** `INITIAL`**).**

- `<<EOF>>` **pour représenter la fin de la source d'entrée.**

- **β**`/`**γ** **pour représenter le mot reconnu par l'ER β si le mot est suivi du mot reconnu par l'ER γ (contexte). Lors de l'analyse, le mot reconnu par l'ER β est retiré de la source d'entrée et le mot suivant reconnu par l'ER γ reste dans la source d'entrée pour la prochaine lecture.**

Par exemple : **β**`$` **est équivalent à β**`/\n`

Par exemple : *Pour l'ER* `"abc"/"123"`*, si on a à l'entrée* `abc12d`*, l'analyseur aboutit à un échec.*

Mais si on a à l'entrée `abc123d` *l'analyseur ne lit que les* `abc` *de l'entrée.*

Attention aux ambiguïtés. Lex les signale.

Par exemple : *Pour l'ER* `[a-zA-Z][a-zA-Z0-9_]*/[a-z]+`*, Lex détecte qu'il y a un problème d'ambiguïté, car les lettres peuvent encore faire partie du mot reconnu à lire.*

- **Priorité décroissante des opérateurs :**
  - `() []`
  - `+ * ?`
  - **produit** ou **concaténation**
  - `|`
  - `/`
  - `^ $`

Par exemple :

```
– abc|def* ≈ abc|(de(f*))
           ≈ (abc)|((de)f*)
           ≈ a(bc)|((de)(f*))
– ab|cd?*  ≈ ab|c(d?)*
           ≈ (ab)|(c((d?)*))
– ^a|bc    ≈ ^(a|bc)
– a|bc$    ≈ (a|bc)$
– a|b/cd   ≈ (a|b)/(cd)
```

- **Lex (ou plutôt sous Linux, Flex pour Fast Lexical Analyzer Generator, voir [2e])**

  - **Fichier d'entrée pour Lex : (nommé généralement avec le suffixe `.lex` ou `.l`)**

    fichier dans lequel on met les règles contenant les ER pour l'analyse lexicale et les actions à exécuter si le mot analysé est correct.

  - **Lex prend un fichier d'entrée et génère, à partir de ce dernier, un fichier C contenant toutes les données et procédures permettant l'analyse lexicale.**

`mon_fichier_dentree.lex` —`lex`→ `lex.yy.c` *(par défaut)*

  - **Format général d'un fichier d'entrée :**

	(I) définitions
%%	
	(II) règles d'analyse et actions
%%	
	(III) codes utilisateur

  - **Dans les parties** *(I)* **et** *(II)***, tous les textes compris entre les bornes `%{` et `%}` sont textuellement et intégralement copiés à la même place dans le fichier C généré. Ces bornes `%{` et `%}` doivent être <u>absolument</u> en début de ligne.**

- *(I)* **<u>définitions</u> :**
  - **On peut y déclarer <u>en début de ligne</u> les conditions inclusives (avec `%s`) et/ou les conditions exclusives (avec `%x`) utilisées (voir *(II)*) pour marquer les règles d'analyse lexicale :**

```
%s ma_cond_incl_1  ma_cond_incl_2
%x ma_cond_excl
%x COMMENTAIRE
```

✓ **Par défaut, seules les règles non marquées (implicitement, elles sont marquées `INITIAL`) sont actives et sont utilisées par l'analyseur.**

✓ **Dans la partie *(II)*, l'instruction `BEGIN(cond);` permet de désactiver la condition précédemment activée et d'activer `cond`.**

✓ **Si la nouvelle condition activée est inclusive, alors toutes les règles non marquées ou marquées par cette condition deviennent actives. Les autres règles deviennent non actives et sont ignorées par l'analyseur.**

✓ **Si la nouvelle condition activée est exclusive, alors seules les règles marquées par cette condition deviennent actives. Les autres règles deviennent non actives et sont ignorées.**

✓ **Pour revenir au condition de départ :** `BEGIN(INITIAL);`
ou `BEGIN(0);`

– **On y définit les ER en les nommant :**

```
ENTIER_RELATIF    -?[0-9]+
IDENTIFICATEUR    [a-zA-Z][a-zA-Z0-9_]*
```

• *(II)* **<u>règles d'analyse et actions</u> :**

– **On y déclare les règles :**

motifs_composés_de_ER ***actions_composées_d'instructions_C***

- **Le motif <u>doit être en début de ligne</u>, sinon la ligne est copiée telle quelle vers le fichier généré (par défaut** `lex.yy.c`**).**

Par exemple :

```
{IDENTIFICATEUR}  { /* blablabla */ }
{ENTIER_RELATIF}  {
                        /* blablabla */
                  }
"/*"                    { BEGIN(COMMENTAIRE); }
<COMMENTAIRE>"*"+"/"    { BEGIN(INITIAL); }
```

- *(III)* **<u>codes utilisateur</u> :**
  - **L'utilisateur définit ici toutes ses procédures et données. Elles seront textuellement et intégralement copiées vers la fin du le fichier C généré.**

- **L'ordre des règles est important.**

  Si un motif est satisfait (c'est à dire que le mot lu en entrée correspond au motif), Lex essaie dans l'ordre les autres motifs des autres règles suivantes :

  - ✓ *S'il n'y a aucun autre motif permettant de lire ce même mot, ce seront les actions associées à ce motif qui seront exécutées.*
  - ✓ *S'il y a un autre motif suivant permettant de lire un mot plus long, Lex le choisira en exécutant les actions qui lui sont associées. Plus exactement, Lex choisit le premier motif permettant de lire le mot le plus long.*

    Attention, pour l'ER avec contexte **β/γ**, *la longueur de* **γ** *est aussi comptée.*
  - ✓ *S'il n'y a que des motifs permettant de lire le même mot (de même longueur donc), Lex choisira les actions du premier motif.*
  - ✓ *On en déduit qu'il faut mettre les motifs particuliers avant les motifs plus généraux.*

Par exemple : (à ne pas faire)

```
-?[0-9]+       { /* blablabla */ }
[0-9]+         { /* on ne passera jamais par ici */ }
```

- **Quelques procédures et variables importantes offertes par Lex :**
  - `yytext` **pointeur sur la chaîne de caractères lus, de type** `char *`
  - `yyleng` **la longueur de la chaîne de caractères lus, de type** `int`
  - `int yylex();` **c'est la procédure principale de l'analyseur lexical, elle lit le mot le plus favorable en entrée et elle doit renvoyer un lexème.**

- **Généralement, la procédure d'analyse lexicale doit retourner deux valeurs :**
  - **un lexème (token) pour indiquer la "chose" lue (par exemple, un entier),**
  - **et la valeur associée à cette "chose" (par exemple, la valeur de l'entier).**
- **Généralement aussi, Lex fonctionne de pair avec Yacc (Yet Another Compiler-Compiler) qui justement, lui fournit la variable**
  - `yylval` **de type** `long` **pour que dans la partie Lex, on puisse lui transmettre cette valeur associée.**

- **Exemple d’utilisation de Lex : fichier d’entrée** `Exemple1.lex`

/* (I) définitions */

```
%{

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

  typedef enum _token {
    TYPE_ENTIER, TYPE_IDENTIFICATEUR, FIN
  } token;

  long yylval;

%}


ENTIER_RELATIF -?[0-9]+
IDENTIFICATEUR [a-zA-Z][a-zA-Z0-9]*
```

```
%%
                    /* (II) règles d'analyse et actions */

[ \t]+ {}

{ENTIER_RELATIF}       {
  yylval = (long)atoi(yytext);
  return(TYPE_ENTIER);
}

{IDENTIFICATEUR} {
  yylval = (long)strdup(yytext);
  return(TYPE_IDENTIFICATEUR);
}

<<EOF>> {
  return(FIN);
}

%%
```

/* (III) codes utilisateur */

```
int main (int argc, char **argv) {
  token i;

  for (;;) {
    switch (yylex()) {
    case TYPE_ENTIER :
      printf("J'ai lu un entier relatif dont la valeur est %d.\n",
             (int)yylval);
      break;
    case TYPE_IDENTIFICATEUR :
      printf("J'ai lu un identificateur de nom %s.\n",
             (char *)yylval);
      break;
    case FIN :
      printf("Il n'y a plus rien à lire.\n");
      return 0;
      break;
    default:;
    }
  }
```

- **Exemple d'utilisation de l'analyseur lexical `lex.yy.c` fourni par Lex et une session avec cet analyseur :**

```
$ lex Exo1.lex
$ gcc -o Exo1 lex.yy.c -ll
$ ./Exo1
ajhgd 1234 a1234 -1234 987ABC
J'ai lu un identificateur de nom ajhgd.
J'ai lu un entier relatif dont la valeur est 1234.
J'ai lu un identificateur de nom a1234.
J'ai lu un entier relatif dont la valeur est -1234.
J'ai lu un entier relatif dont la valeur est 987.
J'ai lu un identificateur de nom ABC.
<Ctrl-D>
Il n'y a plus rien à lire.
$
```

Remarquer comment l'analyseur lexical fourni par Lex analyse le mot `987ABC`.

Fichier d'entrée pour Lex

(I) Définitions :

```
%{
        code de programme C
%}

%s cond_inclusive
%x cond_exclusive

nom1    ER1
...
nomn    ERn
```

(II) Règles d'analyse :

```
%%
ERx    { Action x écrite en C }
...
ERy    { Action y écrite en C }

%%
```

(III) Codes utilisateurs :

instructions C

copie directe

`lex`

copie directe

Fichier C généré par Lex (`lex.yy.c`)

```
code de programme C

extern int    yyleng;
extern char * yytext;
...
int yylex () {
       ...
  while (1) {
       ...
    switch (...) {
      case 1 : { Action x }
         ...
      case n : { Action y }
         ...
    }
  }
}
```

instructions C

- **Avec le mot présent en entrée, Lex essaie dans l'ordre le motif ER de la partie gauche de la 1ère règle.**
- **Puis le motif ER de la partie gauche de la 2ème règle.**
- **… Et ceci jusqu'au motif ER de la partie gauche de la dernière règle.**
- **Plus exactement, Lex cherche à lire le mot le plus long correspondant à un motif et s'arrête au premier caractère qui ne répond plus au motif.**
- **On dit que Lex cherche à lire le mot le plus favorable à un motif ER.**

- **Par exemple, si on a les règles suivantes :**

```
%%
-?[0-9]+      { /* entier relatif avec signe */ }
[a-zA-Z]+     { /* blablabla */ }
[0-9]+        { /* entier naturel sans signe*/ }
%%
```

Et si en entrée, on est en présence du mot `abcd#`**, Lex lit le mot** `abcd` **correspondant à la 2ème règle et s'arrête juste au caractère** `#` **qui restera dans l'entrée pour une prochaine lecture de Lex.**

- **Parmi ces règles, Lex choisira celle qui permet de lire le mot donné en entrée le plus long (favorable).**
- **Et s'il y a plusieurs règles qui permettent de lire le mot donné en entrée le plus long, Lex choisira le premier (dans l'ordre des règles) d'entre elles.**

- **Par exemple :**

```
%%
-?[0-9]+       { /* entier relatif avec signe */ }
[a-zA-Z]+      { /* blablabla */ }
[0-9]+         { /* entier naturel sans signe*/ }
%%
```

Si Lex trouve en entrée le mot `12345a`**, Lex aura le choix entre la règle 1 ou la règle 3, puisque toutes les deux ont des motifs ER acceptant le mot le plus long** `12345`**.**

Lex choisira alors la règle 1. La règle 3 ne sera jamais choisie, elle devient donc inutile. Lex détectera ces règles inutiles avec comme message « `règle non pairée` **».**

- **Il faut donc écrire les règles avec les motifs ER particuliers avant celles avec des motifs ER plus généraux.**
- **Par exemple, si on veut distinguer un entier naturel d'un entier relatif, on doit écrire :**

```
%%
[0-9]+          { /* entier naturel sans signe*/ }
-?[0-9]+        { /* entier relatif avec signe */ }
[a-zA-Z]+       { /* blablabla */ }
%%
```

ou tout simplement :

```
%%
[0-9]+          { /* entier naturel sans signe*/ }
-[0-9]+         { /* entier relatif avec signe */ }
[a-zA-Z]+       { /* blablabla */ }
%%
```

- **En ce qui concerne les conditions sur les motifs ER, un exemple de la déclaration et de l'utilisation :**

```
%s ci1 ci2
%x cx1 cx2 cx3
%s ci3
...
%%
ERa                  { Action a }
<ci3,cx2,ci1>ERb     { Action b }
<cx1,ci3>ERc         { Action c }
<ci2>ERd             { Action d }
...
%%
    code utilisateur
```

avec

`<ci3,cx2,ci1>`***ERb*** `{` ***Action b*** `}` ≈ `<ci3>`***ERb*** `{` ***Action b*** `}`

`<cx2>`***ERb*** `{` ***Action b*** `}`

`<ci1>`***ERb*** `{` ***Action b*** `}`

– **Il faut les voir comme des automates séparés :**
  - **un automate sans condition avec toutes les règles sans condition (par défaut, c'est la condition** `INITIAL`**),**
  - **un automate par condition déclarée, avec toutes les règles marquées par la même condition.**

Ce qui donne pour notre exemple ci-dessus :

ERa { ***Action a*** }

`<cx1>`***ERc*** { ***Action c*** }

`<cx2>`***ERb*** { ***Action b*** }

`<ci1>`***ERb*** { ***Action b*** }

`<ci2>`***ERd*** { ***Action d*** }

`<ci3>`***ERb*** { ***Action b*** }
`<ci3>`***ERc*** { ***Action c*** }

– **Comment ça fonctionne ?**
  - **Par défaut, au départ, seul l'automate sans condition (condition** `INITIAL`**) est activé, les autres sont désactivés (donc ignorés).**

ERa { ***Action a*** }

- **A un moment donné, dans une action, l'instruction** `BEGIN(`**condition**`)` **est exécutée. Cette nouvelle condition est activée.**

  Par exemple : `BEGIN(cx1)`**, une condition** **<u>exclusive</u>** **(**`%x cx1`**), alors seules les règles marquées par cette condition sont activées.**

```
ERa          { Action a }
<cx1>ERc     { Action c }
<cx2>ERb     { Action b }
<ci1>ERb     { Action b }
<ci2>ERd     { Action d }
<ci3>ERb     { Action b }
<ci3>ERc     { Action c }
```

Par exemple : `BEGIN(ci3)`**, une condition** **<u>inclusive</u>** **(**`%s ci3`**), alors seules les règles non marquées et celles marquées par cette condition sont activées.**

```
ERa          { Action a }
<cx1>ERc     { Action c }
<cx2>ERb     { Action b }
<ci1>ERb     { Action b }
<ci2>ERd     { Action d }
<ci3>ERb     { Action b }
<ci3>ERc     { Action c }
```

- **Et pour revenir à la condition initiale** `INITIAL`**, il suffit d'exécuter dans une action** `BEGIN(INITIAL)` **ou** BEGIN(0)**.**

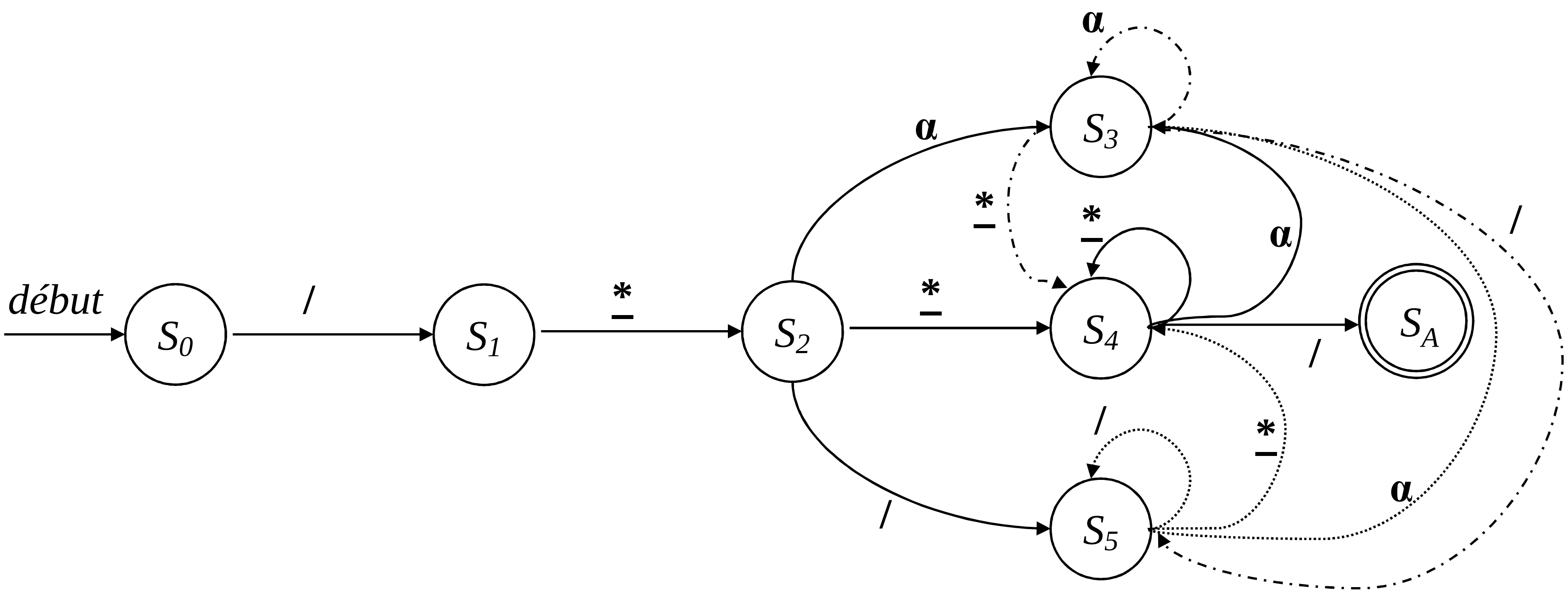
– **Les conditions permettent de simplifier l'écriture des ER en divisant l'ER de départ en plusieurs ER plus simples (diviser pour régner).**

Par exemple, pour les commentaires en C : (en comptant les lignes)

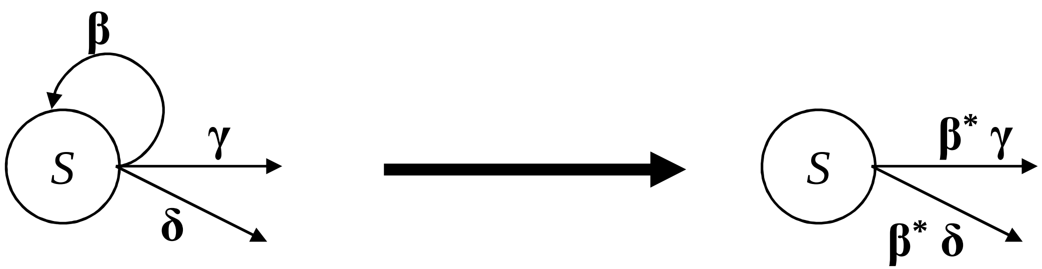
```
%x COMMENTAIRE
...
%%
...
"/*"                          { ...BEGIN(COMMENTAIRE); ... }
<COMMENTAIRE>\n               { .../* on peut compter les lignes*/...}
<COMMENTAIRE>[^*\n]*          { ... }
<COMMENTAIRE>"*"+[^*/\n]*     { ... }
<COMMENTAIRE>"*"+"/"          { ...BEGIN(INITIAL); ... }
...
%%
...
```

– **Sans les conditions, il est difficile de trouver une ER pour ce même travail.**

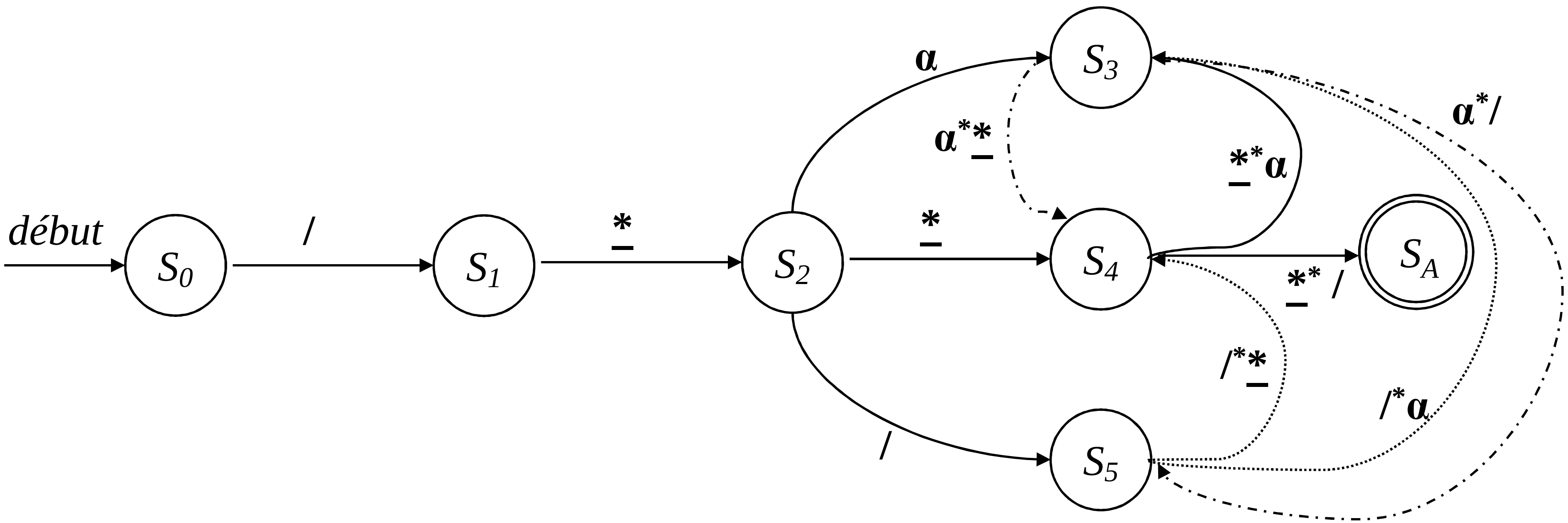
– **Table de transition pour les commentaires en C : (avec α ∈ `[^*/]`)**



– **On enlève chaque arc de fermeture :**

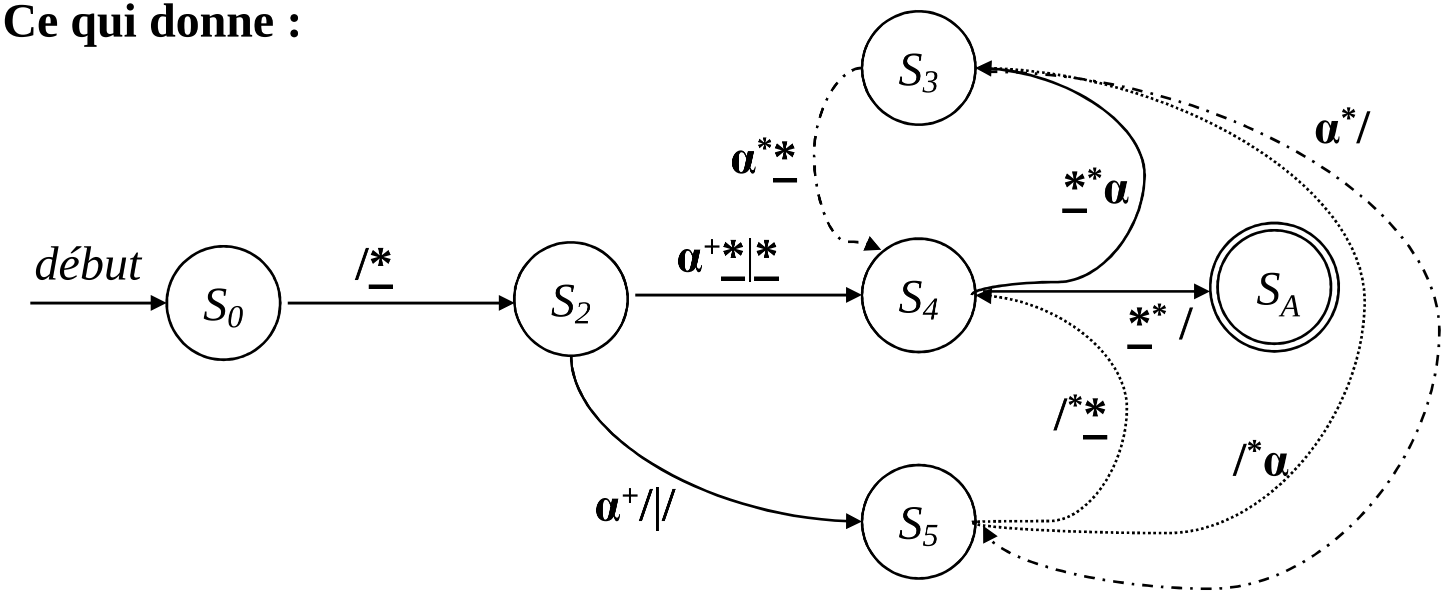


– **Ce qui donne**

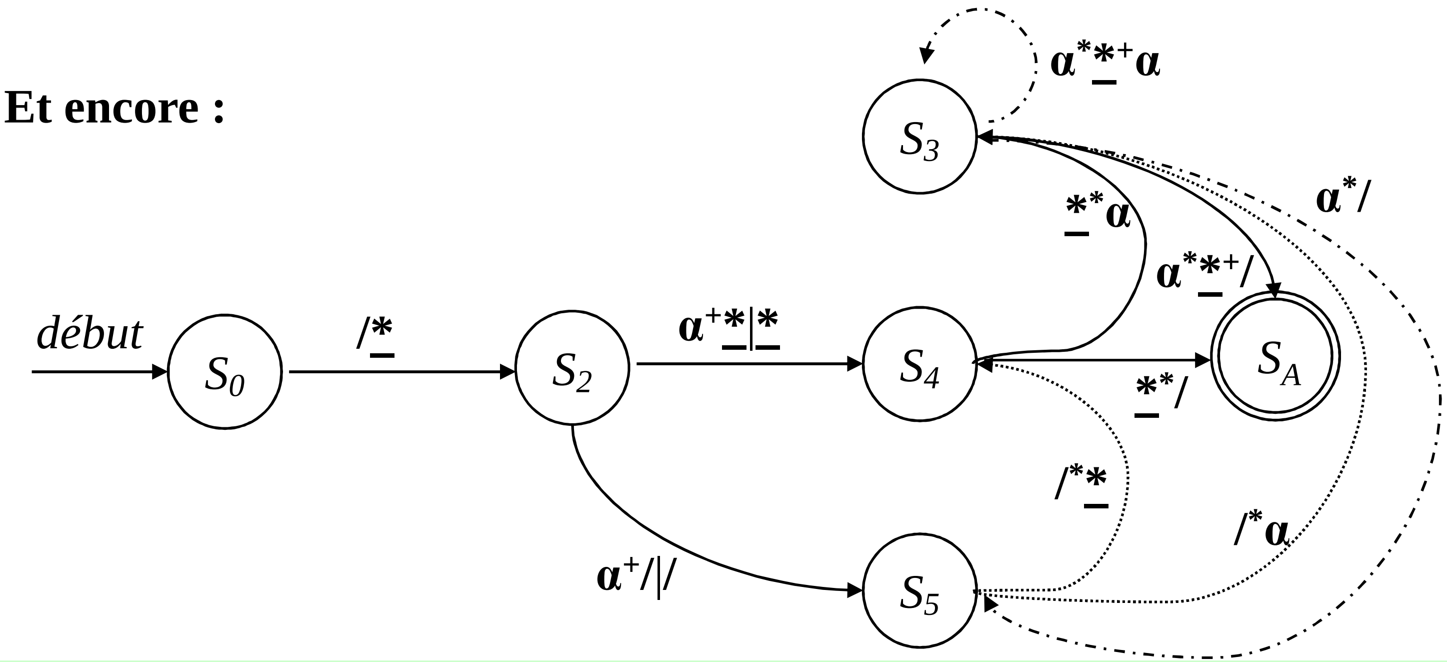


– **On enlève les arcs intermédiaires :**

– **Ce qui donne :**

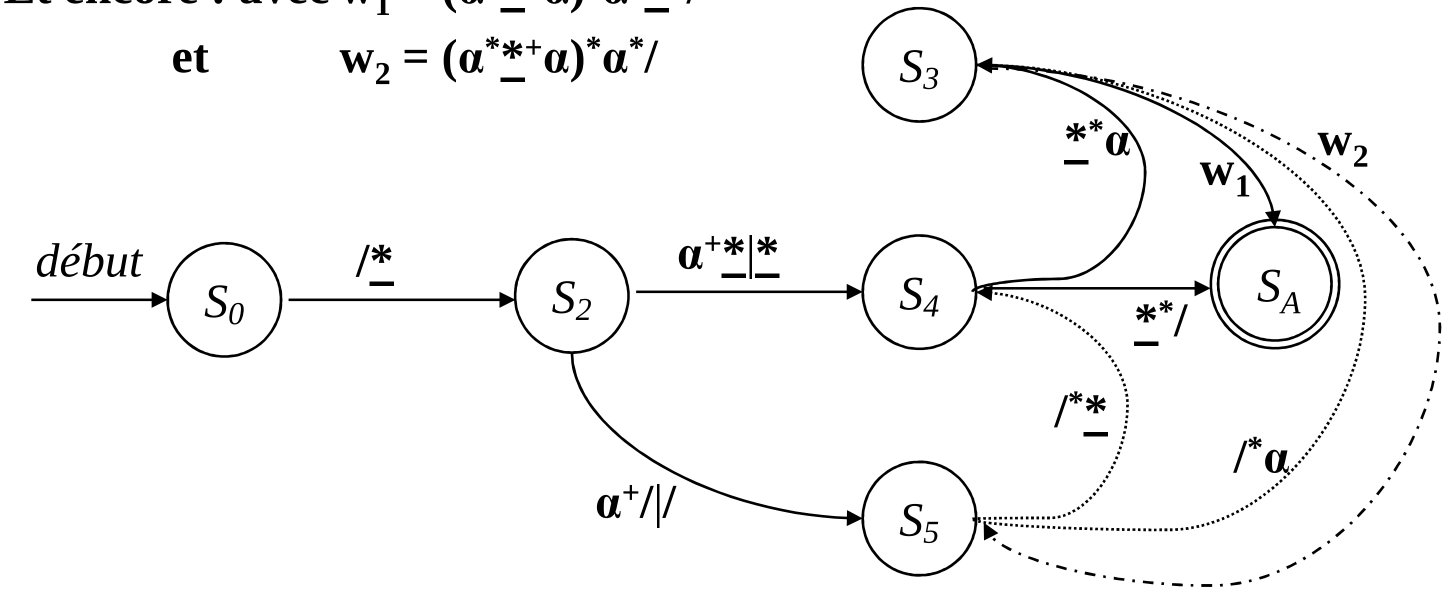


– **Et encore :**



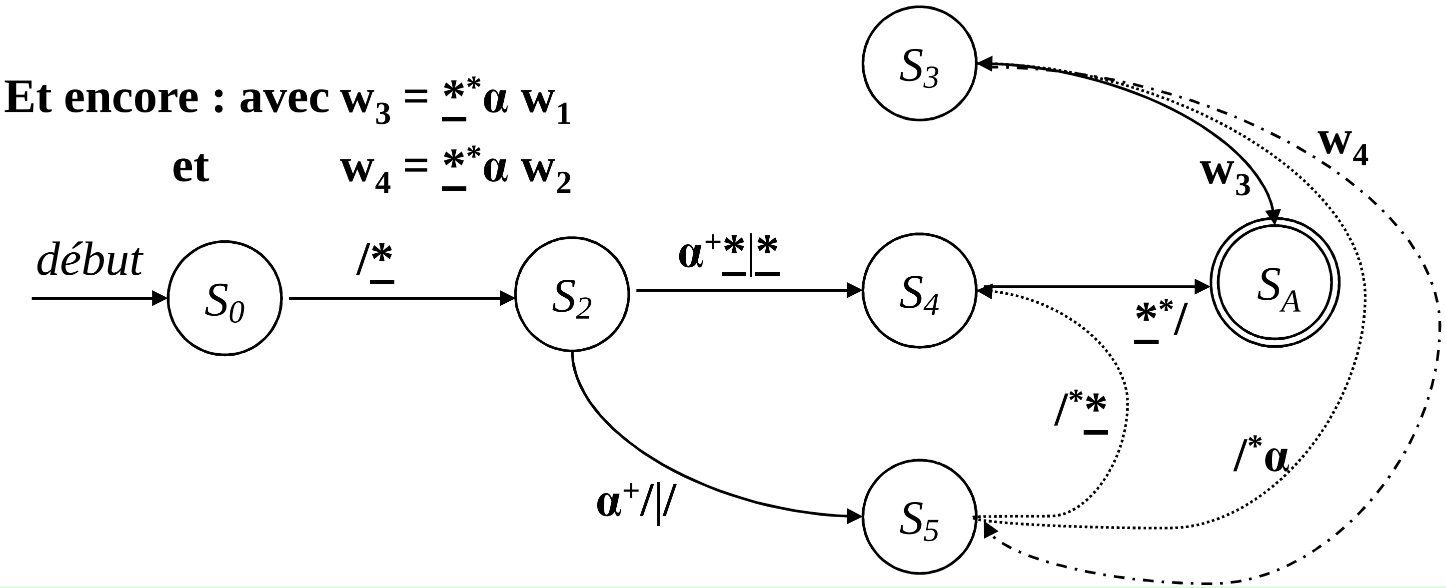
– **Et encore : avec** $\mathbf{w_1 = (\alpha^*\underline{*}^+\alpha)^*\alpha^*\underline{*}^+/}$

et $\mathbf{w_2 = (\alpha^*\underline{*}^+\alpha)^*\alpha^*/}$

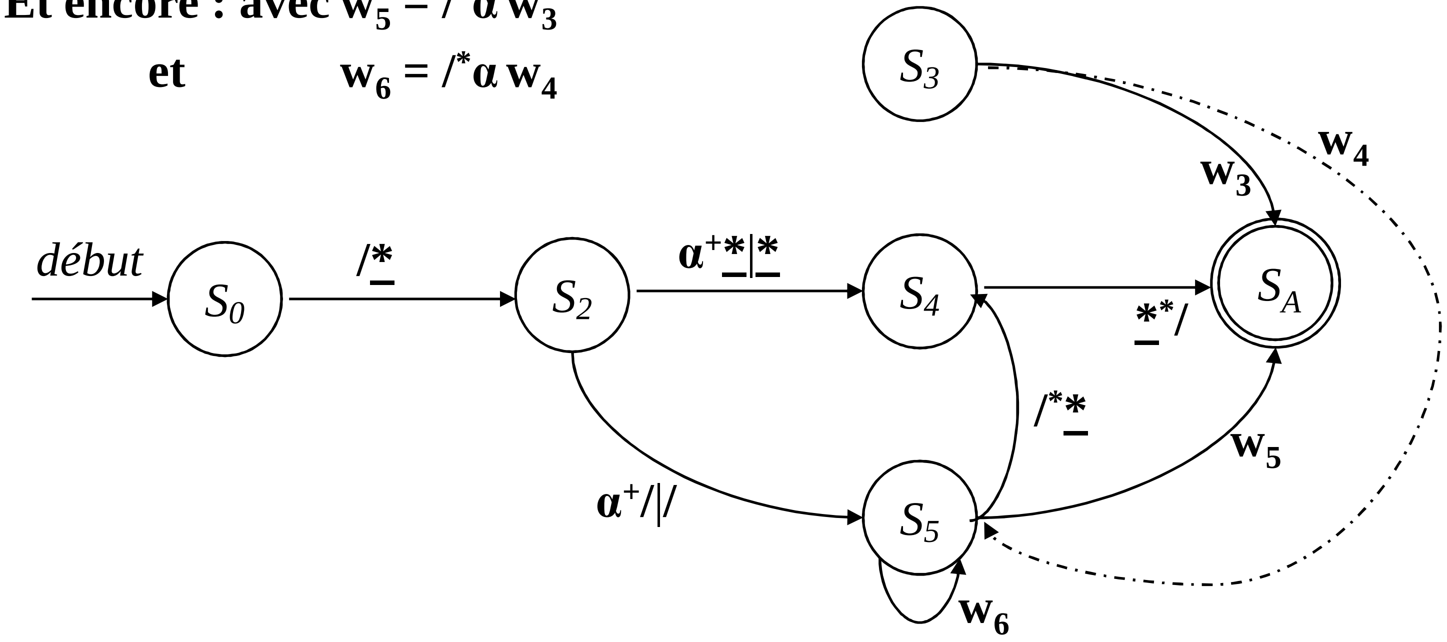


– **Et encore : avec** $\mathbf{w_3 = \underline{*}^*\alpha\ w_1}$

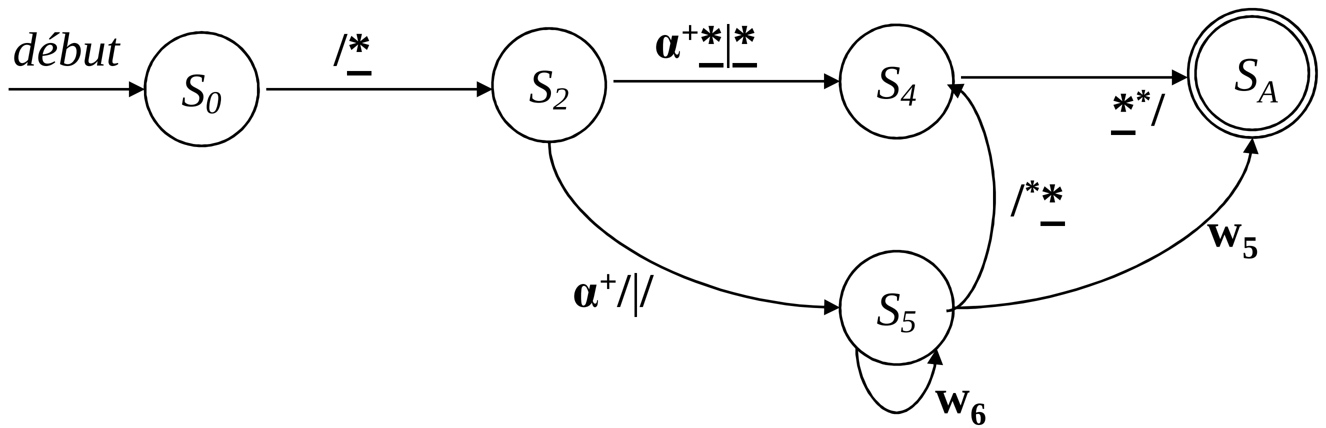
et $\mathbf{w_4 = \underline{*}^*\alpha\ w_2}$



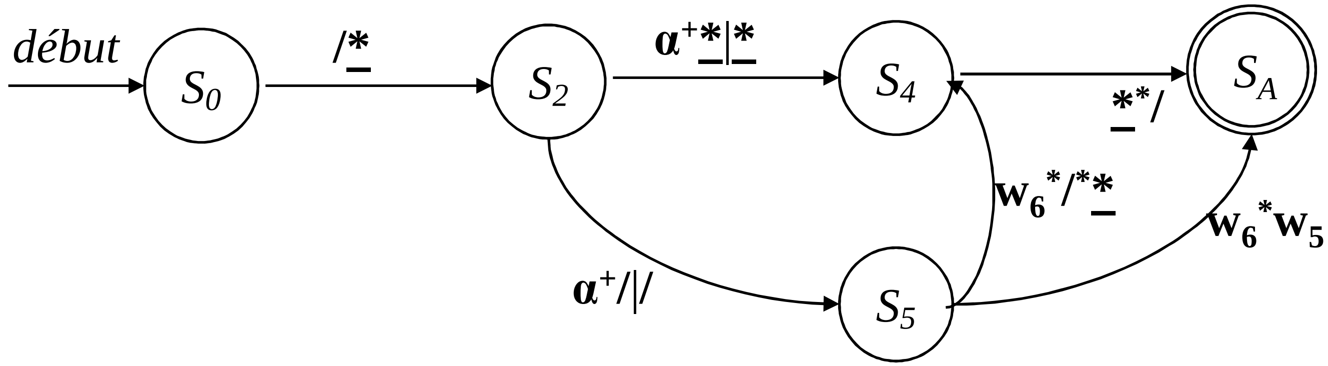
– **Et encore : avec w_5 = /$^*\alpha$ w_3**
et w_6 = /$^*\alpha$ w_4



– **On élimine les états sans arc entrant :**

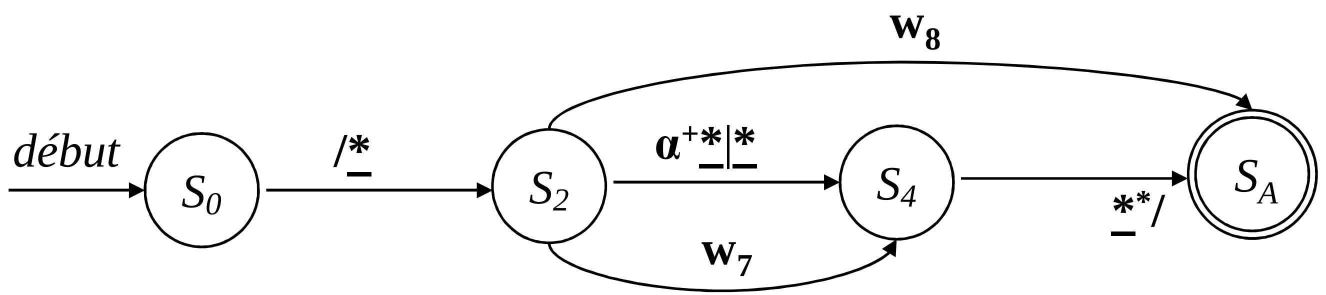


– **Et encore :**

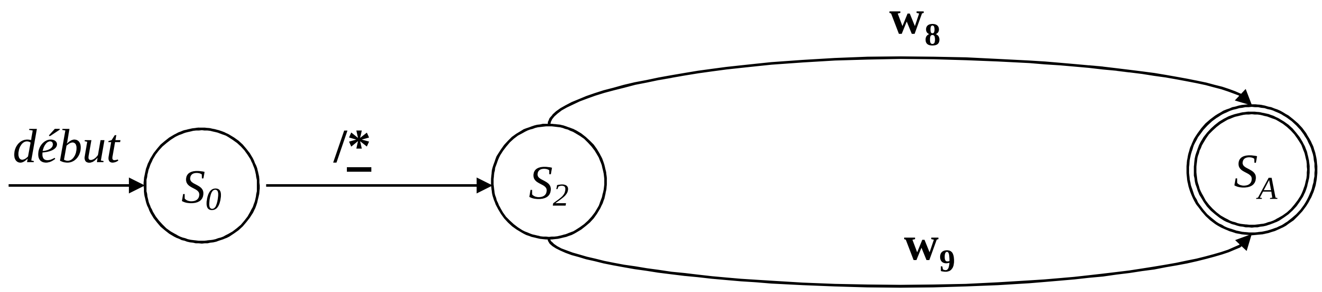


– **Et encore : avec $w_7 = (\alpha^+/|/)\ w_6^*/^*\underline{*}$**

et $w_8 = (\alpha^+/|/)\ w_6^*\ w_5$



– **Et encore : avec $w_9 = (\alpha^{+}\underline{*}|\underline{*}|\ w_7)\ \underline{*}^{*}/$**

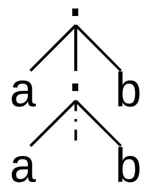


– **Résultat = $/\underline{*}(w_8|w_9)$**

– Et en plus, on ne peut pas compter le nombre de lignes.

- **<u>Limite</u> : par exemple, impossible de trouver les ER correspondant au langage L = { $a^n b^n$ }.**

  On a besoin pour cela une grammaire hors-contexte (non contextuelle).



❑ ocamllex (Générateur d'analyseur lexical pour Objective Caml)

- **Fichier d'entrée pour ocamllex : (nommé généralement avec le suffixe** .mll**)**

  fichier dans lequel on met les règles contenant les ER pour l'analyse lexicale et les actions à exécuter si le mot analysé est correct.

- **Format général d'un fichier d'entrée :**

```
{   en-tête   }                  (* code ocaml entre {} *)

let ident = regexp               (* on définit, si besoin, les ER en les *)
let ident = regexp ...           (*      nommant. Sinon, aucun let *)
rule nom1 arg1... argn =
  parse
        regexp { action }
      | ...
      | regexp { action }
and nom2 arg1... argm =
  parse ...
and ...

{   suite-et-fin   }             (* code ocaml entre {} *)
```

- **{** *en-tête* **}** **même principe que la partie (I) de Lex.**
- **{** *suite-et-fin* **}** **même principe que la partie (III) de Lex.**

- **rule** *nom1* $arg_1 ... arg_n$ **= parse**

  ...

  and *nom2* $arg_1 ... arg_m$ **= parse**

  ...

sera tranformé par ocamllex en

let rec *nom1* $arg_1 ... arg_n$ **lexbuf =**

...

and *nom2* $arg_1 ... arg_m$ **lexbuf =**

...

- $arg_1 ... arg_n$ **et** $arg_1 ... arg_m$ **peuvent être vide.**
- **lexbuf a été ajouté et correspond au buffer d'entrée où vont lire les fonctions** nom_1, nom_2, *etc,...*

- **Expression rationnelle (étendue) : (ER pour ocamllex)**
  - **`‘x‘`** **pour représenter un caractère x quelconque.**
  - `_` **(souligné) pour représenter n’importe quel caractère.**
  - `eof` **pour représenter la fin de l’entrée.**
  - `[‘a‘ ‘b‘-‘e‘ ‘\n‘ ‘5‘-‘9‘]` **ensemble de caractères**
  - `[^ ‘a‘ ‘b‘-‘e‘ ‘\n‘ ‘5‘-‘9‘]` **tous sauf ceux-ci-dessus.**
  - β? $\varepsilon|\beta$
  - β* $\varepsilon|\beta|\beta\beta|\beta\beta\beta|\beta\beta\beta\beta|\ldots$
  - β+ $\beta|\beta\beta|\beta\beta\beta|\beta\beta\beta\beta|\ldots$
  - β # γ **pour représenter l’ensemble des caractères de β qui n’appartient pas à γ (β et γ sont des ensembles de caractères ci-dessus).**
  - $\beta\ \gamma$ **pour le produit (concaténation)**
  - β | γ **pour l’union (choix)**
  - ($\beta\gamma\delta$) **pour pouvoir regrouper les ER β, γ et δ.**
  - *ident* **défini par `let`.**
  - β `as` *ident* **si le motif β est satisfait, le résultat (type `string`) de la lecture se trouve dans *ident*.**
  - **Précédence décroissante : `*` et `+`, `?`, concaténation, `|`, `as`**

- **ocamllex prend un fichier d'entrée et génère, à partir de ce dernier, un fichier** ml **contenant toutes les données et procédures permettant l'analyse lexicale.**

`mon_fichier_dentree.mll` $\xrightarrow{\texttt{ocamllex}}$ `mon_fichier_dentree.ml`

- **fichier d'entrée** `Exemple1_ocamllex.mll` **: (voir** `Exemple1.lex`**)**

```
{                          (* (I) *)

open Lexing;;
open Printf;;
open String;;

type token =
     TYPE_ENTIER of int
   | TYPE_IDENTIFICATEUR of string
   | FIN
;;

}
```

```
                      (* (II) *)
let ENTIER_RELATIF = '-'?['0'-'9']+
let IDENTIFICATEUR = ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9']*
let SEPARATEUR    = [' ' '\n' '\t']

rule analyseur_lexical = parse
     SEPARATEUR+       { analyseur_lexical lexbuf }
   | (ENTIER_RELATIF as x) SEPARATEUR
        { TYPE_ENTIER (int_of_string x) }
   | (IDENTIFICATEUR as x) SEPARATEUR { TYPE_IDENTIFICATEUR x }
   | eof               { FIN }

{                     (* (III) *)

let buffer_entree = from_channel stdin
in while true do
     match analyseur_lexical buffer_entree with
       TYPE_ENTIER x ->
         printf
           "J'ai lu un entier relatif dont la valeur est %d.\n" x;
         flush stdout
     | TYPE_IDENTIFICATEUR x ->
         printf "J'ai lu un identificateur de nom %s.\n" x;
         flush stdout
```

```
    | FIN ->
        printf "Il n'y a plus rien à lire.\n";
        exit 0
    done;;
}
```

- **Exemple d'utilisation de l'analyseur lexical fourni par ocamllex et une session avec cet analyseur :**

```
$ ocamllex Exemple1_camllex.mll
$ ocamlc -o Exemple1_camllex Exemple1_camllex.ml
$ ./Exemple1_camllex.ml
ajhgd 1234 a1234 -1234 987ABC
J'ai lu un identificateur de nom ajhgd.
J'ai lu un entier relatif dont la valeur est 1234.
J'ai lu un identificateur de nom a1234.
J'ai lu un entier relatif dont la valeur est -1234.
J'ai lu un entier relatif dont la valeur est 987.
J'ai lu un identificateur de nom ABC.
<Ctrl-D>
Il n'y a plus rien à lire.
$
```

❑ Genlex (module)

- **Outils pour créer des analyseurs lexicaux simples et standards (et particulièrement adaptés pour le langage OCaml).**
- **La fonction** `make_lexer` **prend une liste de mots clefs (donnés sous forme de chaîne de caractères** `string`**) et retourne un analyseur lexical qui**
  - **prend un flot de caractères en entrée,**
  - **reconnaît et traite les mots clefs donnés ci-dessus, identificateurs, entiers, flottants, chaînes de caractères, caractères et les commentaires OCaml (en ignorant ces derniers) :** `"blablabla" , 'a' , (* blablabla *)`
  - **rend un flot de token (lexème) correspondant (voir ci-dessous).**
- **On y trouve le type somme** `token` **avec ses constructeurs :**

```
type token =
    Kwd    of string
  | Ident  of string
  | Int    of int
  | Float  of float
  | String of string
  | Char   of char
;;
```

- Exemple1_genlex.ml

```
#load "camlp4o.cma";;

let keywords =
 [ "REM"; "GOTO"; "LET"; "PRINT"; "INPUT"; "IF"; "THEN";
   "~"; "!"; "+"; "-"; "*"; "/"; "%";
   "="; "<"; ">"; "<="; ">="; "<>";
   "&"; "|" ];;

let line_lexer l = Genlex.make_lexer keywords (Stream.of_string l);;

let rec foo s =
  match s with parser
    [< 'Genlex.Kwd x >] -> "clef" :: foo s
  | [< 'Genlex.Ident x >] -> "identificateur" :: foo s
  | [< 'Genlex.Int x >] -> "entier" :: foo s
  | [< 'Genlex.Float x >] -> "flottant" :: foo s
  | [< 'Genlex.String x >] -> "chaine" :: foo s
  | [< 'Genlex.Char x >] -> "caractere" :: foo s
  | [<>] -> []
;;
```

```
let rec fii = parser
    [< 'Genlex.Kwd x ; y = fii >] -> "clef" :: y
  | [< 'Genlex.Ident x ; y = fii >] -> "identificateur" :: y
  | [< 'Genlex.Int x ; y = fii >] -> "entier" :: y
  | [< 'Genlex.Float x ; y = fii >] -> "flottant" :: y
  | [< 'Genlex.String x; y = fii >] -> "chaine" :: y
  | [< 'Genlex.Char x ; y = fii >] -> "caractere" :: y
  | [<>] -> []
;;


foo (line_lexer "LET (* blablabla *) x = x + y * 3");;
       ➔
["clef"; "identificateur"; "clef"; "identificateur"; "clef";
 "identificateur"; "clef"; "entier"]


fii (line_lexer "LET x = x + y * (* blablabla *) 3");;
       ➔
["clef"; "identificateur"; "clef"; "identificateur"; "clef";
 "identificateur"; "clef"; "entier"]
```

3. Analyse syntaxique

- **Syntaxe concrète :**
  - **Une suite de caractères**

    qui forment des mots (lexèmes, voir l'analyse lexicale ci-dessus)

    qui forment des phrases (expressions ou instructions)

    qui forment un programme.
  - **Par exemple :**

```
fact := 1;
i := 1;
TANTQUE i <= n FAIRE {
  fact := fact * i;
  i := i + 1
};
ECRIRE "La valeur factorielle de n = ";
ECRIRE fact;
ECRIRE "\n";
```

- **Les mots se composent entre eux en suivant des règles formulées dans une grammaire hors-contexte (non contextuelle, context-free ou BNF pour Backus-Naur Form) : (**voir **MiniLangage**, un peu plus loin**)**
- **Par exemple :**

SEQUENCE-INSTRUCTIONS ➔ *INSTRUCTION-SIMPLE*
| *INSTRUCTION-SIMPLE* **;** *SEQUENCE-INSTRUCTIONS*

INSTRUCTION-SIMPLE ➔ …
| *AFFECTATION*
| *ECRITURE*
| *BOUCLE-TANTQUE*
| …

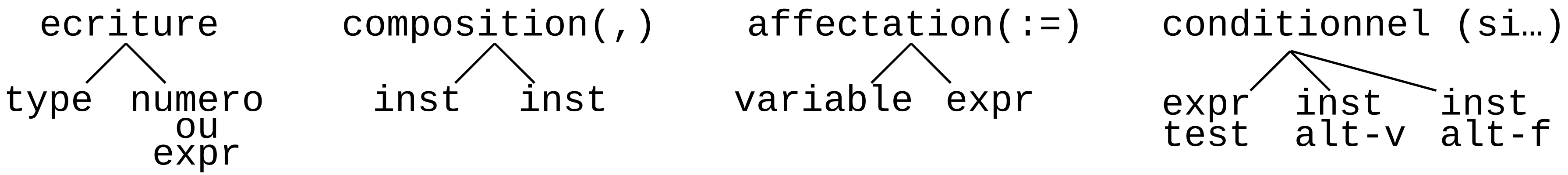
- **La grammaire permet de lier la syntaxe concrète à la syntaxe abstraite (voir ci-dessous).**

• **Syntaxe abstraite :**

- **Une représentation (généralement arborescente) de la structure des expressions, instructions et programme.**
- **Permettant de représenter la structure en machine.**
- **Le choix est fortement dirigé par la grammaire.**

- **Par exemple : (syntaxe abstraite présentée par des structures C)**

```
typedef struct _INST_ARBRE {
  enum INST_TYPE type;
  union {
      ...
    struct {
      enum ECRITURE_TYPE type;
      union {
        int numero;
        struct _EXPR_ARBRE *expr;
      } forme;
    } ecriture;
    struct {
      struct _INST_ARBRE *instg;
      struct _INST_ARBRE *instd;
    } composition;
    struct {
      char *variable_droite;
      struct _EXPR_ARBRE *expr_gauche;
    } affectation;
    struct {
      struct _EXPR_ARBRE *expr_test;
      struct _INST_ARBRE *altern_v;
      struct _INST_ARBRE *altern_f;
    } conditionnel;
    struct {
      struct _EXPR_ARBRE *expr_test;
      struct _INST_ARBRE *corps;
    } boucle;
  } forme;
} *INST_ARBRE;
```



- **Par exemple : (en OCaml)**

```
type expr_arbre =
     ...
and forme_ecriture =
     Numero of int
   | Expression of expr_arbre
and inst_arbre =
     ...
   | Ecriture of forme_ecriture
   | Composition of inst_arbre * inst_arbre
   | Affectation of string * expr_arbre
   | Conditionnel of expr_arbre * inst_arbre * inst_arbre
   | Boucle of expr_arbre * inst_arbre
;;
```

- **<u>Grammaire</u> : <T, N, P, S>**
  - **T** un ensemble fini de symboles terminaux (mots du langage),
  - **N** un ensemble fini de symboles non-terminaux,
  - ***P* : *v* → *w*** un ensemble de règles de production où
    - v et w sont des mots formés de terminaux et/ou de non-terminaux,
    - w peut être vide (noté ε pour le mot vide).
  - S un axiome (non-terminal). Tout mot du langage est produit à partir de l'axiome.

- **<u>Grammaire non-contextuelle (context-free, hors-contexte ou BNF pour Backus-Naur Form)</u> :**

  - ***P* : *v* → *w*** un ensemble de règles de production où v est un-non terminal et où w est vide (noté ε) ou un mot formé de terminaux et/ou de non-terminaux,

Par exemple : < {a, b}, {S}, {S→aSb ; S→ε}, S >

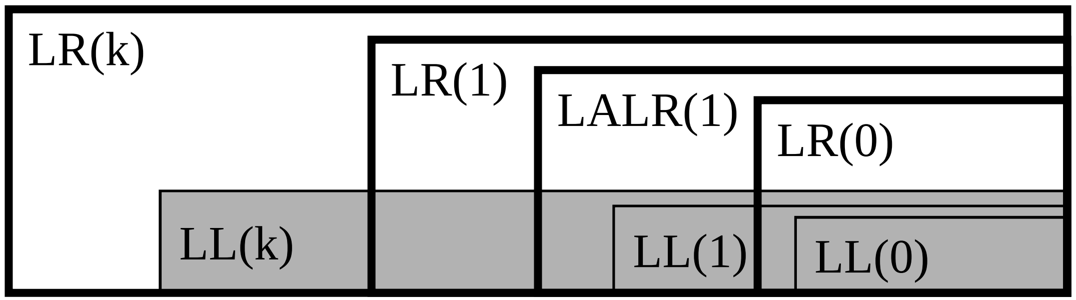
- **Un langage non-contextuel est un langage généré par une grammaire non-contextuelle.**
- **Tous les langages de programmation (C, Fortran, Lisp, Pascal, Java, etc, …) sont non-contextuels. Plus exactement, chacun est spécifié par une grammaire non-contextuelle.**
- **Par exemple : (langage C)**

…

Instruction →
`if (`***Expression***`)` ***Instruction***
| `if (`***Expression***`)` ***Instruction*** `else` ***Instruction***
| `while (`***Expression***`)` ***Instruction***
| …

Expression →
Expression Opérateur-binaire Expression
| …
| `++` ***lvalue***
| …

- **Deux familles remarquables encore et toujours utilisées : LL(1) et LR(1) (en particulier LALR(1), sous-famille du LR(1)).**
- **1 veut dire un symbole lu en avance pour pouvoir déterminer la suite de l'analyse : LL(k) nécessite k symboles lus en avance.**
- **LL(1) : L (Left-to-right scanning of the input) et L (Leftmost derivation) avec une analyse descendante.**
- **LR(1) : L (Left-to-right scanning of the input) et R (constructing Rightmost derivation in reverse) avec une analyse ascendante.**
- **LALR(1) : LA (LookAhead) et LR(1) comme ci-dessus.**



Une démonstration est donnée dans :

On the relationship between the LL(k) and LR(k) grammar

(Anton Nijholt - Information Processing Letters, Volume 15, Issue 3)

- **LL(1) : Analyse prédictive descendante**
  - **Calcul du** `Premier(X)` **:**
    - **Si X∈T (ensemble des symboles terminaux),**

      `Premier(X)` **= { X }.**
    - **Si X∈N (ensemble des symboles non-terminaux)**
      - **et si il y a une règle X → ε, on ajoute ε dans** `Premier(X)`
      - **et si il y a une règle** $X \rightarrow Y_1 Y_2 \ldots Y_k$ **(avec** $Y_i \in (T \cup N)$ **), on ajoute** `a` **(∈T) dans** `Premier(X)` **s'il existe i tel que** `a` **est dans** `Premier(`Y_i`)` **<u>et que</u> ε est dans tous les** `Premier(`Y_1`)`**, …,** `Premier(`Y_{i-1}`)`**.**

      Si ε est dans `Premier(`Y_j`)`**, pour tout j = 1, 2, …, k, alors on ajoute ε dans** `Premier(X)`**.**

    - **Par abus, on étend le calcul à** `Premier(β)` **avec** $\beta = Z_1 Z_2 \ldots Z_n$ **(avec** $Z_i \in (T \cup N)$ **), on ajoute** `a` **(∈T) dans** `Premier(β)` **s'il existe i tel que** `a` **est dans** `Premier(`Z_i`)` **<u>et que</u> ε est dans tous les** `Premier(`Z_1`)`**,** `Premier(`Z_2`)`**, …,** `Premier(`Z_{i-1}`)`**.**

      Si ε est dans `Premier(`Z_j`)`**, pour tout j = 1, 2, …, n, alors on ajoute ε dans** `Premier(β)`**.**
    - `Premier()` **⊂ (T ∪{ε}).**

- `Premier()` **est un ensemble qui se calcule parfois en y ajoutant les éléments des autres** `Premier()`**.**
- **Ce calcul peut se croiser (récursivement) :**

  `Premier(`**X**`)` **se calcule à partir de** `Premier(`**Y**`)` **qui se calcule à partir de** `Premier(`**Z**`)` **… qui se calcule à partir du** `Premier(`**X**`)`**.**
- **Mais ce calcul par ajout se termine nécessairement, car T (ensemble des terminaux) et N (ensemble des non-terminaux) sont finis.**

• **Calcul du** `Suivant(`**Y**`)` **: (seulement pour les non-terminaux Y∈N)**

- **Mettre** $ **(marqueur de fin de la source d'entrée à analyser) dans** `Suivant(`**S**`)` **où S est l'axiome.**
- **S'il y a une production A → αBβ, le contenu de** `Premier(`**β**`)`**, excepté ε, est ajouté dans** `Suivant(`**B**`)`**.**
- **S'il existe une production A → αB ou A → αBβ telle que** `Premier(`**β**`)` **contient ε, les éléments de** `Suivant(`**A**`)` **sont ajoutés à** `Suivant(`**B**`)`**.**
- `Suivant()` **est un ensemble qui se calcule en y ajoutant les éléments des** `Premier()` **et/ou des** `Suivant()` **des autres.**
- **Même remarque que pour** `Premier()` **: le calcul se termine.**
- `Suivant()`$\subset$ **(T ∪{$}).** **<u>Attention</u> : ε ∉** `Suivant()`.

- **Par exemple, pour la grammaire : (avec E comme axiome)**

E → T E’
E’ → + T E’ | ε
T → F T’
T’ → * F T’ | ε
F → (E) | c_ou_v

- **Ce qui donne :**

`Premier(`**E**`)` = `Premier(`**T**`)` = `Premier(`**F**`)` = **{ (, c_ou__v }**
`Premier(`**E’**`)` = **{ +, ε }**
`Premier(`**T’**`)` = **{ *, ε }**
`Suivant(`**E**`)` = `Suivant(`**E’**`)` = **{), $ }**
`Suivant(`**T**`)` = `Suivant(`**T’**`)` = **{ +,), $ }**
`Suivant(`**F**`)` = **{ +, *,), $ }**

- **On construit la table d'analyse prédictive :**

  1. **Pour chaque production A → α de la grammaire, procéder aux étapes 2 et 3 :**
  2. **Pour chaque terminal x dans `Premier(α)`,**

     ajouter A → α à la table M[A, x].
  3. **Si ε est dans `Premier(α)`, ajouter A → α à M[A, y] pour chaque terminal y dans `Suivant(A)`. Si ε est dans `Premier(α)` et $ est dans `Suivant(A)`, ajouter A → α à M[A, $].**

- **Si chaque entrée de la table est prise au plus par une règle, alors la grammaire est LL(1).**

- **On peut démontrer que les règles de production d'une grammaire LL(1) ne peuvent pas être**
  - **de récursivité gauche :**

    A → Aα   ou   A → … → Aα
  - **à facteur gauche commun : (ci-dessous un exemple avec deux règles concernant A avec x comme facteur gauche commun)**

    A → xα

    A → xβ

- **Ce qui donne, pour notre exemple, la table M suivante :**

	c_ou_v	+	*	(	)	$
E	**E→TE'**			**E→TE'**		
E'		**E'→+TE'**			**E'→ε**	**E'→ε**
T	**T→FT'**			**T→FT'**		
T'		**T'→ε**	**T'→*FT'**		**T'→ε**	**T'→ε**
F	**F→c_ou_v**			**F→(E)**		

- **La grammaire est donc LL(1).**

- **L'analyse de la source d'entrée avec la table se fait par :**
  - **On empile \$, on empile ensuite le symbole de l'axiome.**
  - **Répéter**

    Soit X le symbole en sommet de pile et a le symbole repéré à la source d'entrée,

    Si X est un terminal ou \$ alors

    Si X = a alors

    enlever X de la pile et avancer la tête de lecture

    sinon erreur

    sinon X est un non-terminal,

    Si M[X, a] = $X \rightarrow Y_1Y_2\ldots Y_k$ alors

    enlever X de la pile;

    empiler Y_k, puis Y_{k-1}, …, puis enfin Y_1

    sinon erreur

  Jusqu'à X = \$

- **Par exemple :**

Pile	Entrée
$E	cv + cv * cv $
$E'T	cv + cv * cv $
$E'T'F	cv + cv * cv $
$E'T'cv	cv + cv * cv $
$E'T'	+ cv * cv $
$E'	+ cv * cv $
$E'T+	+ cv * cv $
$E'T	cv * cv $

$E'T'F	cv * cv $
$E'T'cv	cv * cv $
$E'T'	* cv $
$E'T'F*	* cv $
$E'T'F	cv $
$E'T'cv	cv $
$E'T'	$
$E'	$
$	$

- **La phrase de la source d'entrée** cv + cv * cv $ **est donc acceptée (correcte).**

- **D’une manière générale, pour le LL(1), on peut voir chaque non-terminal comme une procédure récursive :**

  E → T E’

  E’ → + T E’ | ε

  T → F T’

  T’ → * F T’ | ε

  F → (E) | c_ou_v

```
procedure E :
  si car_à_lire == c_ou_v ou car_à_lire == ‘(’
  alors appel de T;       //résultat de T dans la pile
        appel de E’;      //résultat de E’ dans la pile
        on dépile E’, puis T;
        on forme E→TE’ et on l’empile
  sinon ERREUR
fin procedure E;
```

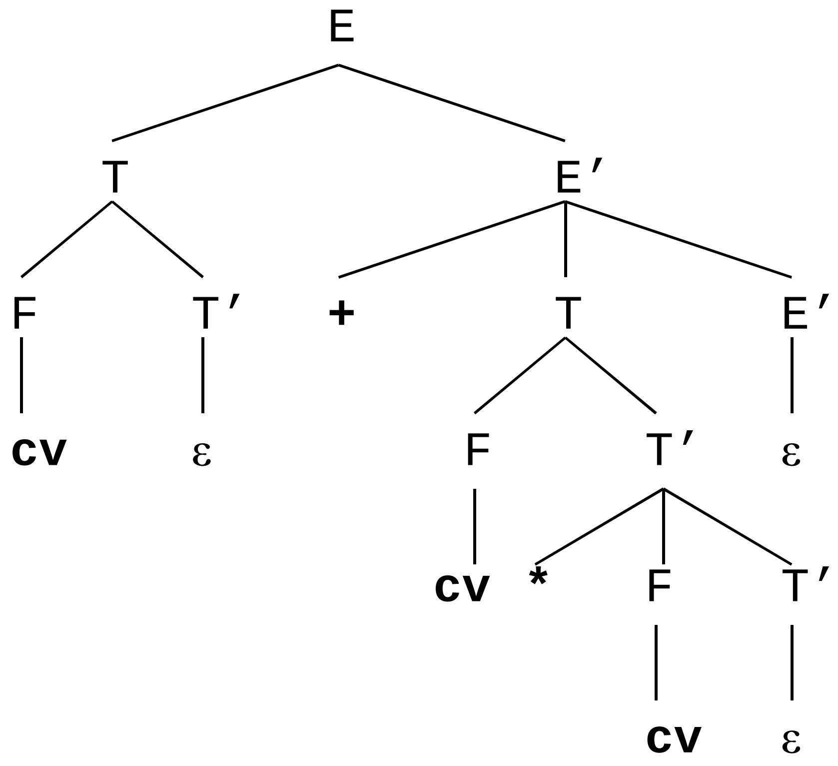
```
procedure E' :
  si car_à_lire == '+'
  alors lire '+'
        appel de T;       //résultat de T dans la pile
        appel de E';      //résultat de E' dans la pile
        on dépile E', puis T;
        on forme E'→+TE' et on l'empile
  sinon si car_à_lire == ')' ou car_à_lire == '$'
  alors on forme E'→ε et on l'empile
  sinon ERREUR
fin procedure E';
```

```
procedure T :
  si car_à_lire == c_ou_v ou car_à_lire == '('
  alors appel de F;       //résultat de F dans la pile
        appel de T';      //résultat de T' dans la pile
        on dépile T', puis F;
        on forme T→FT' et on l'empile
  sinon ERREUR
fin procedure T;
```

```
procedure T’ :
  si car_à_lire == ‘*’
  alors lire ‘*’
        appel de F;      //résultat de F dans la pile
        appel de T’;     //résultat de T’ dans la pile
        on dépile T’, puis F;
        on forme T’→*FT’ et on l’empile
  sinon si car_à_lire == ‘+’ ou car_à_lire == ‘)’
             ou car_à_lire == ‘$’
  alors on forme T’→ε et on l’empile
  sinon ERREUR
fin procedure T’;
```

```
procedure F :
  si car_à_lire == c_ou_v
  alors lire c_ou_v;
        on forme F→c_ou_v et on l’empile
  sinon si car_à_lire == ‘(‘
  alors lire ‘(‘; appel E; lire ‘)’; //résultat de E dans la pile
        on dépile E; on forme F→(E) et on l’empile
  sinon ERREUR
fin procedure F;
```

- **En reprenant la phrase précédente** cv + cv * cv $ **, et en appelant la procédure E (axiome), nous avons :**



- **Quelques définitions et règles concernant les expressions arithmétiques :**

  - **en notation préfixée : + * 4 5 + x y**

    E → + E E | * E E | c_ou_v (LL(1))

  - **en notation postfixé : 4 5 * x y + +**

    E → E E + | E E * | c_ou_v (non LL(1))

  - **en notation infixe complètement parenthésée : ((4 * 5) + (x + y))**

    E → (E + E) | (E * E) | c_ou_v (non LL(1))

    E → (E O E) | c_ou_v (LL(1))
    O → + | *

    E → T F | c_ou_v (LL(1))
    T → (E
    F → + E) | * E)

– **en notation infixe traditionnelle avec priorité des opérateurs :**

4 * 5 + (x + y)

E → E + E | E * E | (E) | c_ou_v (non LL(1), ni LR(1), ni aucun autre, … car ambiguë)

E → E + T | T (non LL(1))
T → T * F | F
F → (E) | c_ou_v

E → T E' (LL(1) en supprimant la récursivité gauche)
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | c_ou_v

❑ LR(1) et LALR(1) : Analyse ascendante

- **Elément (appelé aussi item) LR(1) :**
  - **une grammaire** <T, N, P, S>**,**
  - **une règle de la grammaire :** $X \rightarrow \alpha\beta$
    avec α et β des mots appartenant à $(T \cup N)^*$.

    Par exemple, pour la grammaire des expressions arithmétiques traditionnelles :

    T = { `i`**,** `+`**,** `-`**,** `*`**,** `/` **}** `i` *représentant les variables et constantes*

    N = { E, T, F **}**

    P :
    ```
    E → E + T | E – T | T
    T → T * F | T / F | F
    F → ( E ) | i
    ```
    S = E
  - **élément :** $[X \rightarrow \alpha|\beta, a]$ **(avec $a \in T$)**
  - **avec un seul symbole terminal a, d'où le 1 de LR(1).**
- **Etat = ensemble de ces éléments.**

- **Il faut voir l'élément [X→ α|β, a]**
  - **comme un outil permettant d'analyser la source donnée en entrée,**
  - **selon la règle X → αβ,**
  - **avec la tête de lecture juste positionnée devant le mot β,**
  - **après avoir déjà lu le mot α,**
  - **reste à lire le mot β,**
  - **une fois que le mot αβ sera lu, on devra tomber sur le terminal a en entrée.**

- **Par exemple : [E→E|+T, $]**
  - **selon la règle E → E+T,**
  - **avec la tête de lecture juste positionnée devant le symbole terminal +,**
  - **après avoir déjà lu une expression E,**
  - **reste à lire le symbole terminal +, suivi d'une expression T,**
  - **une fois que E+T sera lu, on devra tomber sur le symbole $ en entrée ($ utilisé comme symbole marquant la fin de la source d'entrée.**

- <u>**Fermeture d'un état I**</u>
  - **J = I**
  - **répéter**
    - **pour chaque item [A→α|Bβ, a] de J, chaque production B→γ de G' et chaque terminal b de** `Premier(`**βa**`)` **telle que [B→|γ, b] n'est pas dans J, ajouter [B→|γ, b] à J**

    jusqu'à ce qu'aucun autre item ne puisse être ajouté à J;
  - **résultat J**

- **Par exemple : I = { [E→E|+T, $] }**

  `Fermeture(`**I**`)` **= I, car + est un symbole terminal**

- **Par exemple : I = { [E→|E+T,$] }**

  `Fermeture(`**I**`)` **= { [E→|E+T, $], [E→|E+T, +], [E→|T, +], … }**

- **<u>Transition depuis un ensemble d'items I sur le symbole Y</u>**
  - **Y est un terminal ou non terminal,**
  - **repérer les éléments de I de la forme [A →α|Yβ, a],**
  - **rasssembler dans un ensemble J tous ces éléments en avançant "|" vers la droite d'un symbole Y ➔ J = { …, [A →αY|β, a], …}**
  - **résultat** `Fermeture(`**J**`)`**.**

- **On note aussi** `Transition(`**I, Y**`)` **par I.Y .**

- **Par exemple : I = { [X→|E, $], [E→|E+T, $], [E→|E+T, +], [E→|T, +], … }**
  `Transition(`**I, E**`)` **= { [X→E|, $], [E→E|+T, $], [E→E|+T, +] }**

- **Par exemple : I = { …, [F→|(E),$], [F→|(E),$], [F→|(E),$], …}**
  `Transition(`**I, (** `)` **= { [F→(|E+T, $], [F→(|E),$], [F→(|E),$],**
  [E→|E+T, +], [E→|T, +], … }

Construction des états (ensembles fermés d'items)

- **Augmenter la grammaire G en G' avec la règle S'→S où S est l'axiome de G et avec le caractère $ comme marqueur de fin.**
- ε_0 = `Fermeture(`**{ [S'→|S, $] }**`);`
- **Répéter**
  - **pour chaque état ε et pour chaque symbole x (x ∈ (T ∪ N)) tel que**

    `Transition(`**ε, x**`)` **soit non vide et ne corresponde pas à un état existant; Nommer** ε_i = `Transition(`**ε, x**`)` **où i est un nouvel indice.**

  jusqu'à ce qu'aucun nouvel état ne puisse plus être créé.

- **On appelle l'ensemble de ces états ainsi trouvés la collection canonique d'ensembles d'items LR(1) pour la grammaire augmentée de G.**

- **Par exemple, avec la grammaire dont la règle de production est**

  E → E **+** T | T

  T → T ***** F | F

  F → **(** E **)** | **i**

On obtient : avec X le nouvel axiome, E l'ancien et le marqueur de fin $

ε_0 = { [X→|E,$], [E→|E+T,$], [E→|T,$], [E→|E+T,+], [E→|T,+], [T→|T*F,$], [T→|F,$], [T→|T*F,+], [T→|F,+], [T→|T*F,*], [T→|F,*], [F→|(E),$], [F→|i,$], [F→|(E),+], [F→|i,+], [F→|(E),*], [F→|i,*] }

ε_0 . (= ε_1 = { [F→(|E),$], [E→|E+T,)], [E→|T,)], [E→|E+T,+], [E→|T,+], [T→|T*F,)], [T→|F,)], [T→|T*F,+], [T→|F,+], [T→|T*F,*], [T→|F,*], [F→|(E),)], [F→|i,)], [F→|(E),+], [F→|i,+], [F→|(E),*], [F→|i,*], [F→(|E),+], [F→(|E),*] }

ε_0 . E = ε_2 = { [X→E|,$], [E→E|+T,$], [E→E|+T,+] }

ε_0 . F = ε_3 = { [T→F|,$], [T→F|,+], [T→F|,*] }

ε_0 . T = ε_4 = { [E→T|,$], [E→T|,+], [T→T|*F,$], [T→T|*F,+], [T→T|*F,*] }

ε_0 . i = ε_5 = { [F→i|,$], [F→i|,+], [F→i|,*] }

ε_1 . (= ε_6 = { [F→(|E),)], [E→|E+T,)], [E→|T,)], [E→|E+T,+], [E→|T,+], [T→|T*F,)], [T→|F,)], [T→|T*F,+], [T→|F,+], [T→|T*F,*], [T→|F,*], [F→|(E),)], [F→|i,)], [F→|(E),+], [F→|i,+], [F→|(E),*], [F→|i,*], [F→(|E),+], [F→(|E),*] }

ε_1 . E = ε_7 = { [F→(E|),$], [E→E|+T,)], [E→E|+T,+], [F→(E|),+], [F→(E|),*] }

ε_1 . F = ε_8 = { [T→F|,)], [T→F|,+], [T→F|,*] }

ε_1 . T = ε_9 = { [E→T|,)], [E→T|,+], [T→T|*F,)], [T→T|*F,+], [T→T|*F,*] }

ε_1 . i = ε_{10} = { [F→i|,)], [F→i|,+], [F→i|,*] }

ε_2 . + = ε_{11} = { [E→E+|T,$], [T→|T*F,$], [T→|F,$], [T→|T*F,*], [T→|F,*],
[F→|(E),$], [F→|i,$], [F→|(E),*], [F→|i,*], [E→E+|T,+],
[T→|T*F,+], [T→|F,+], [F→|(E),+], [F→|i,+] }

ε_4 . * = ε_{12} = { [T→T*|F,$], [F→|(E),$], [F→|i,$], [T→T*|F,+], [F→|(E),+],
[F→|i,+], [T→T*|F,*], [F→|(E),*], [F→|i,*] }

ε_6 . (= ε_6 = { [F→(|E),)], [E→|E+T,)], [E→|T,)], [E→|E+T,+], [E→|T,+],
[T→|T*F,)], [T→|F,)], [T→|T*F,+], [T→|F,+], [T→|T*F,*],
[T→|F,*], [F→|(E),)], [F→|i,)], [F→|(E),+], [F→|i,+],
[F→|(E),*], [F→|i,*], [F→(|E),+], [F→(|E),*] }

ε_6 . E = ε_{13} = { [F→(E|),)], [E→E|+T,)], [E→E|+T,+], [F→(E|),+], [F→(E|),*] }

ε_6 . F = ε_8 = { [T→F|,)], [T→F|,+], [T→F|,*] }

ε_6 . T = ε_9 = { [E→T|,)], [E→T|,+], [T→T|*F,)], [T→T|*F,+], [T→T|*F,*] }

ε_6 . i = ε_{10} = { [F→i|,)], [F→i|,+], [F→i|,*] }

ε_7 .) = ε_{14} = { [F→(E)|,$], [F→(E)|,+], [F→(E)|,*] }

ε_7 . + = ε_{15} = { [E→E+|T,)], [T→|T*F,)], [T→|F,)], [T→|T*F,*], [T→|F,*], [F→|(E),)], [F→|i,)], [F→|(E),*], [F→|i,*], [E→E+|T,+], [T→|T*F,+], [T→|F,+], [F→|(E),+], [F→|i,+] }

ε_9 . * = ε_{16} = { [T→T*|F,)], [F→|(E),)], [F→|i,)], [T→T*|F,+], [F→|(E),+], [F→|i,+], [T→T*|F,*], [F→|(E),*], [F→|i,*] }

ε_{11} . (= ε_1 = { [F→(|E),$], [E→|E+T,)], [E→|T,)], [E→|E+T,+], [E→|T,+], [T→|T*F,)], [T→|F,)], [T→|T*F,+], [T→|F,+], [T→|T*F,*], [T→|F,*], [F→|(E),)], [F→|i,)], [F→|(E),+], [F→|i,+], [F→|(E),*], [F→|i,*], [F→(|E),+], [F→(|E),*] }

ε_{11} . F = ε_3 = { [T→F|,$], [T→F|,+], [T→F|,*] }

ε_{11} . T = ε_{17} = { [E→E+T|,$], [T→T|*F,$], [T→T|*F,*], [E→E+T|,+], [T→T|*F,+] }

ε_{11} . i = ε_5 = { [F→i|,$], [F→i|,+], [F→i|,*] }

ε_{12} . (= ε_1 = { [F→(|E),$], [E→|E+T,)], [E→|T,)], [E→|E+T,+], [E→|T,+], [T→|T*F,)], [T→|F,)], [T→|T*F,+], [T→|F,+], [T→|T*F,*], [T→|F,*], [F→|(E),)], [F→|i,)], [F→|(E),+], [F→|i,+], [F→|(E),*], [F→|i,*], [F→(|E),+], [F→(|E),*] }

ε_{12} . F = ε_{18} = { [T→T*F|,$], [T→T*F|,+], [T→T*F|,*] }

ε_{12} . i = ε_5 = { [F→i|,$], [F→i|,+], [F→i|,*] }

ε_{13} .) = ε_{19} = { [F→(E)|,)], [F→(E)|,+], [F→(E)|,*] }

ε_{13} . + = ε_{15} = { [E→E+|T,)], [T→|T*F,)], [T→|F,)], [T→|T*F,*], [T→|F,*], [F→|(E),)], [F→|i,)], [F→|(E),*], [F→|i,*], [E→E+|T,+], [T→|T*F,+], [T→|F,+], [F→|(E),+], [F→|i,+] }

ε_{15} . (= ε_{6} = { [F→(|E),)], [E→|E+T,)], [E→|T,)], [E→|E+T,+], [E→|T,+], [T→|T*F,)], [T→|F,)], [T→|T*F,+], [T→|F,+], [T→|T*F,*], [T→|F,*], [F→|(E),)], [F→|i,)], [F→|(E),+], [F→|i,+], [F→|(E),*], [F→|i,*], [F→(|E),+], [F→(|E),*] }

ε_{15} . F = ε_{8} = { [T→F|,)], [T→F|,+], [T→F|,*] }

ε_{15} . T = ε_{20} = { [E→E+T|,)], [T→T|*F,)], [T→T|*F,*], [E→E+T|,+], [T→T|*F,+] }

ε_{15} . i = ε_{10} = { [F→i|,)], [F→i|,+], [F→i|,*] }

ε_{16} . (= ε_{6} = { [F→(|E),)], [E→|E+T,)], [E→|T,)], [E→|E+T,+], [E→|T,+], [T→|T*F,)], [T→|F,)], [T→|T*F,+], [T→|F,+], [T→|T*F,*], [T→|F,*], [F→|(E),)], [F→|i,)], [F→|(E),+], [F→|i,+], [F→|(E),*], [F→|i,*], [F→(|E),+], [F→(|E),*] }

ε_{16} . F = ε_{21} = { [T→T*F|,)], [T→T*F|,+], [T→T*F|,*] }

ε_{16} . i = ε_{10} = { [F→i|,)], [F→i|,+], [F→i|,*] }

ε_{17} . * = ε_{12} = { [T→T*|F,$], [F→|(E),$], [F→|i,$], [T→T*|F,+], [F→|(E),+], [F→|i,+], [T→T*|F,*], [F→|(E),*], [F→|i,*] }

ε_{20} . * = ε_{16} = { [T→T*|F,)], [F→|(E),)], [F→|i,)], [T→T*|F,+], [F→|(E),+], [F→|i,+], [T→T*|F,*], [F→|(E),*], [F→|i,*] }

- **<u>Construction de la table d'analyse LR(1)</u>**

1. **Construire la collection des ensembles d'items LR(1) pour la grammaire augmentée;**
2. **Si [A→α|aβ, b] est dans ε_i et** `Transition(`**ε_i, a**`)` **= ε_j, remplir Action[i, a] avec *"décaler j" (shift)*. Ici a doit être un terminal.**

   Si [A→α|, a] est dans ε_i, remplir Action[i, a] avec

   ***"réduire par A→ α" (reduce)*. Ici A ne doit pas être l'axiome S'.**

   Si [S'-->S|, $] est dans ε_i, remplir Action[i, $] avec *"accepter"*
3. **On construit les transitions Successeur pour tout non terminal A en utilisant la règle** :

   Si `Transition(`**ε_i, A**`)` = **ε_j, alors Successeur**[i, **A**] = j.
4. **Toutes les entrées non définies par les règles (2) et (3) sont positionnées à *"erreur"*.**
5. **L'état initial est ε_0.**

- **<u>Définition</u> : LR(1)**

  On dit que la grammaire augmentée est LR(1) si les règles se trouvant dans (2) n’engendrent pas des actions conflictuelles de type *décaler*/*réduire* ou *réduire*/*réduire*.

– **Par exemple, avec la grammaire précédente, on obtient :**

	(	)	*	+	E	F	T	i	$
E0	D1				S2	S3	S4	D5	
E1	D6				S7	S8	S9	D10	
E2				D11					A
E3			R4	R4					R4
E4			D12	R2					R2
E5			R6	R6					R6
E6	D6				S13	S8	S9	D10	
E7		D14		D15					
E8		R4	R4	R4					
E9		R2	D16	R2					
E10		R6	R6	R6					
E11	D1					S3	S17	D5	
E12	D1					S18		D5	
E13		D19		D15					
E14			R5	R5					R5
E15	D6					S8	S20	D10	
E16	D6					S21		D10	
E17			D12	R1					R1
E18			R3	R3					R3
E19		R5	R5	R5					
E20		R1	D16	R1					
E21		R3	R3	R3					

- **L'analyse :**

- **ε_0** dans la pile et initialiser le pointeur source **ps** sur le premier symbole du mot donné en entrée **w$**
- **Répéter indéfiniement**

début
. Soit **e** l'état en sommet de pile et **a** le symbole pointé par **ps**;
. **Si** Action[**e**,**a**] = décaler vers **e'**
alors **début**
. empiler **a** puis **e'**; **(1)**
. avancer **ps** d'un symbole
fin
sinon si Action[**e**,**a**] = réduire par **A→b**
alors début
. dépiler 2 fois la longueur du mot **b**; **(2)**
. soit **e'** le nouvel état au sommet de la pile;
. empiler **A** puis Successeur[**e'**,**A**]; **(3)**
. émettre une action en sortie correspondant à cette réduction :
- construire l'arbre (ici, ci-dessous notre exemple)
- ou écrire en sortie la règle **A→b**
- ou ... ou ne rien écrire
fin
sinon si Action[**e**,**a**] = accepter
alors retourner (le mot d'entrée est correct)
sinon erreur.
fin

- **En notant**

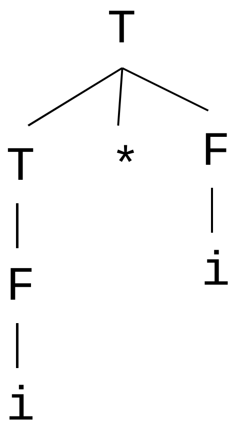
```
{F→i} ≈  F
         |
         i
```

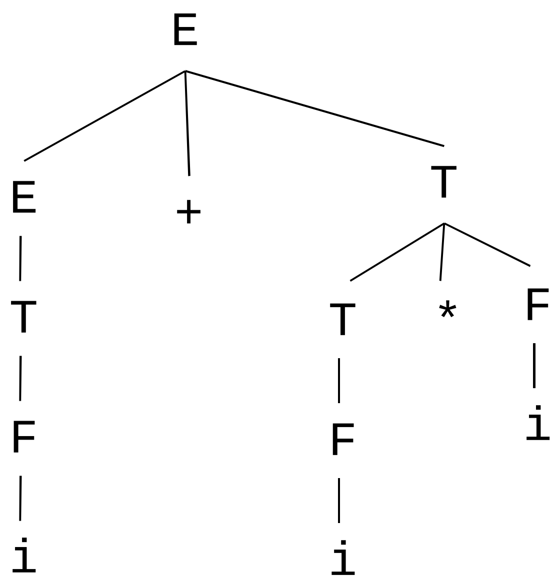
```
{T→F→i} ≈  T
           |
           F
           |
           i
```

```
{E→T→F→i} ≈  E
             |
             T
             |
             F
             |
             i
```

{T→{T→F→i}{*}{F→i}}

≈





≈

```
{ E→{E→T→F→i}
    {+}
    {T→{T→F→i}
       {*}
       {F→i}} }
```

- **Exemple :**

pile d'analyse (sens de l'empilement ➔)	Entrée
ε_0	**i** + i * i $
en appliquant (1), ce qui donne	
ε_0 {i} ε_5	+ i * i $
ε_0 {i} ε_5	**+** i * i $
en appliquant (2), ce qui donne	
ε_0	+ i * i $
puis en appliquant (3), ce qui donne	
ε_0 {<u>F</u>➔i} ε_3	+ i * i $
ε_0 {F➔i} ε_3	**+** i * i $
en appliquant (2), ce qui donne	
ε_0	+ i * i $
puis en appliquant (3), ce qui donne	
ε_0 {<u>T</u>➔F➔i} ε_4	+ i * i $

ε_0 {T➔F➔i} ε_4	+ i * i $
ε_0 {T➔F➔i} $\boldsymbol{\varepsilon_4}$	**+** i * i $
en appliquant (2), ce qui donne	
ε_0	+ i * i $
puis en appliquant (3), ce qui donne	
ε_0 {$\underline{\text{E}}$➔T➔F➔i} ε_2	+ i * i $
ε_0 {E➔T➔F➔i} $\boldsymbol{\varepsilon_2}$	**+** i * i $
en appliquant (1), ce qui donne	
ε_0 {E➔T➔F➔i} ε_2 {+} ε_{11}	i * i $
ε_0 {E➔T➔F➔i} ε_2 {+} $\boldsymbol{\varepsilon_{11}}$	**i** * i $
en appliquant (1), ce qui donne	
ε_0 {E➔T➔F➔i} ε_2 {+} ε_{11} {i} ε_5	* i $
en appliquant (2), ce qui donne	
ε_0 {E➔T➔F➔i} ε_2 {+} ε_{11}	* i $

puis en appliquant (3), ce qui donne	
ε_0 {E→T→F→i} ε_2 {+} ε_{11} {$\underline{\mathrm{F}}$→i} ε_3	$\ast\ i\ \$$
ε_0 {E→T→F→i} ε_2 {+} ε_{11} {F→i} $\boldsymbol{\varepsilon_3}$	$\boldsymbol{\ast}\ i\ \$$
en appliquant (2), ce qui donne	
ε_0 {E→T→F→i} ε_2 {+} ε_{11}	$\ast\ i\ \$$
puis en appliquant (3), ce qui donne	
ε_0 {E→T→F→i} ε_2 {+} ε_{11} {$\underline{\mathrm{T}}$→F→i} ε_{17}	$\ast\ i\ \$$
ε_0 {E→T→F→i} ε_2 {+} ε_{11} {$\underline{\mathrm{T}}$→F→i} $\boldsymbol{\varepsilon_{17}}$	$\boldsymbol{\ast}\ i\ \$$
en appliquant (1), ce qui donne	
ε_0 {E→T→F→i} ε_2 {+} ε_{11} {T→F→i} ε_{17} {*} ε_{12}	$i\ \$$
ε_0 {E→T→F→i} ε_2 {+} ε_{11} {T→F→i} ε_{17} {*} $\boldsymbol{\varepsilon_{12}}$	$\mathbf{i}\ \$$
en appliquant (1), ce qui donne	
ε_0 {E→T→F→i} ε_2 {+} ε_{11} {T→F→i} ε_{17} {*} ε_{12} {i} ε_5	$\$$

ε_0 {E➔T➔F➔i} ε_2 {+} ε_{11} {T➔F➔i} ε_{17} {*} ε_{12} {i} $\boldsymbol{\varepsilon_5}$	**$**
en appliquant (2), ce qui donne	
ε_0 {E➔T➔F➔i} ε_2 {+} ε_{11} {T➔F➔i} ε_{17} {*} ε_{12}	$
puis en appliquant (3), ce qui donne	
ε_0 {E➔T➔F➔i} ε_2 {+} ε_{11} {T➔F➔i} ε_{17} {*} ε_{12} {**F**➔i} ε_{18}	$
ε_0 {E➔T➔F➔i} ε_2 {+} ε_{11} {T➔F➔i} ε_{17} {*} ε_{12} {F➔i} $\boldsymbol{\varepsilon_{18}}$	**$**
en appliquant (2), ce qui donne	
ε_0 {E➔T➔F➔i} ε_2 {+} ε_{11}	$
puis en appliquant (3), ce qui donne	
ε_0 {E➔T➔F➔i} ε_2 {+} ε_{11} {**T**➔{T➔F➔i}{*}{F➔i}} ε_{17}	$
ε_0 {E➔T➔F➔i} ε_2 {+} ε_{11} {**T**➔{T➔F➔i}{*}{F➔i}} $\boldsymbol{\varepsilon_{17}}$	**$**
en appliquant (2), ce qui donne	
ε_0	$

puis en appliquant (3), ce qui donne	
ε_0 {E➔{E➔T➔F➔i}{+}{T➔{T➔F➔i}{*}{F➔i}}**} ε_2**	$
ε_0 {E➔{E➔T➔F➔i}{+}{T➔{T➔F➔i}{*}{F➔i}}**} ε_2**	**$**
accepter	

- **<u>Définition</u> :**

Pour tout item [A→α|β, b], il y a deux composants :

- le premier composant est A→α|β

- le second composant est b.

On appelle le cœur d'un état l'ensemble des premiers composants de ses items.

- **<u>Construction de la table d'analyse LALR(1)</u>**

1. **Construire la collection des ensembles d'items LR(1) pour la grammaire augmentée;**
2. **Pour chaque cœur présent parmi les états, trouver tous les états ayant ce même cœur et remplacer ces états par leur union.**
3. **Dans la nouvelle collection résultante, tester les trois conditions de la règle (2) de la construction de la table d'analyse LR(1) (voir ci-dessus). Si ces conditions conduisent à un conflit, la grammaire n'est pas LALR(1).**
4. **La fonction Transition est construite comme suit :**

   Soit un état J de la nouvelle collection ($J = I_1 \cup I_2 \cup ... I_n$, avec I_i de l'ancienne collection) et soit K l'union de tous les états ayant le même cœur que Transition(J,X), alors Transition(J, X) = K .
5. **Les actions d'analyse Action et Successeur de l'état ε_i comme dans Construction de la table d'analyse LR(1) (voir ci-dessus).**

– **Par exemple, avec la grammaire LR(1) précédente dont les règles de production est**

E → E **+** T | T

T → T ***** F | F

F → **(** E **)** | **i**

– **On obtient :**

Les fusions :

E1-6	E12-16
E3-8	E14-19
E4-9	E17-20
E5-10	E18-21
E7-13	
E11-15	

La table : pas de conflit ➔ la grammaire est LALR(1)

	(	)	*	+	E	F	T	i	$
E0	D1-6				S2	S3-8	S4-9	D5-10	
E1-6	D1-6				S7-13	S3-8	S4-9	D5-10	
E2				D11-15					A
E3-8		R4	R4	R4					R4
E4-9		R2	D12-16	R2					R2
E5-10		R6	R6	R6					R6
E7-13		D14-19		D11-15					
E11-15	D1-6					S3-8	S17-20	D5-10	
E12-16	D1-6					S18-21		D5-10	
E14-19		R5	R5	R5					R5
E17-20		R1	D12-16	R1					R1
E18-21		R3	R3	R3					R3

- **<u>Cas LR(1) qui n'est pas LALR(1)</u>**

S→>Aa|bAc|Bc|bBa
A→d
B→d

On obtient :

ε_0 = { [X→|S,$], [S→|Aa,$], [S→|bAc,$], [S→|Bc,$], [S→|bBa,$], [A→|d,a], [B→|d,c] }

ε_0 . A = ε_1 = { [S→A|a,$] }

ε_0 . B = ε_2 = { [S→B|c,$] }

ε_0 . S = ε_3 = { [X→S|,$] }

ε_0 . b = ε_4 = { [S→b|Ac,$], [A→|d,c], [S→b|Ba,$], [B→|d,a] }

ε_0 . d = ε_5 = { [A→d|,a], [B→d|,c] }

ε_1 . a = ε_6 = { [S→Aa|,$] }

ε_2 . c = ε_7 = { [S→Bc|,$] }

ε_4 . A = ε_8 = { [S→bA|c,$] }

ε_4 . B = ε_9 = { [S→bB|a,$] }

ε_4 . d = ε_{10} = { [A→d|,c], [B→d|,a] }

ε_8 . c = ε_{11} = { [S→bAc|,$] }

ε_9 . a = ε_{12} = { [S→bBa|,$] }

La table : pas de conflit ➔ la grammaire est LR(1)

	A	B	S	a	b	c	d	$
E0	S1	S2	S3		D4		D5	
E1				D6				
E2						D7		
E3								A
E4	S8	S9					D10	
E5				R5		R6		
E6								R1
E7								R3
E8						D11		
E9				D12				
E10				R6		R5		
E11								R2
E12								R4

La fusion : ε_5 et ε_{10}

La table après fusion : conflit ➔ la grammaire n'est pas LALR(1)

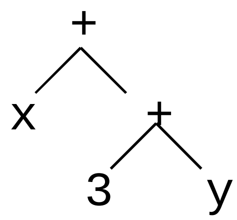
	A	B	S	a	b	c	d	$
E0	S1	S2	S3		D4		D5-10	
E1				D6				
E2						D7		
E3								A
E4	S8	S9					D5-10	
E5-10				**R5R6**		**R6R5**		
E6								R1
E7								R3
E8						D11		
E9				D12				
E11								R2
E12								R4

- **<u>En résumé</u>,**
  - ✓ **Au moment de choisir, l'analyseur LR(1) offre plus de choix que l'analyseur LL(1) : chaque état regroupe plusieurs items dont chacun exprime un choix possible.**
  - ✓ **Ce qui entraîne beaucoup d'états**
    - **→ analyseur plus gros, plus lourd et plus gourmand.**
  - ✓ **Un compromis (si possible) est fait par l'analyseur LALR(1) en mettant ensemble (si possible) les états qui font la même chose au caractère attendu près. On dit qu'ils ont le même cœur.**
  - ✓ **Résultat (si possible)**
    - **→ moins d'états,**
    - **→ analyseur plus petit, plus léger et moins gourmand.**

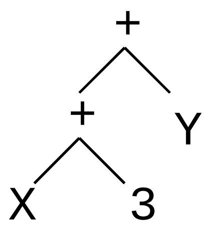
- **<u>Cas d'ambiguïté</u> : (plus ou moins résolu par Yacc)**

  ✓ **E → E + E | …**

  x + 3 + y →

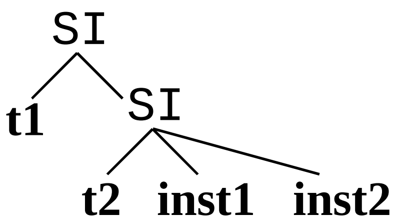


OU

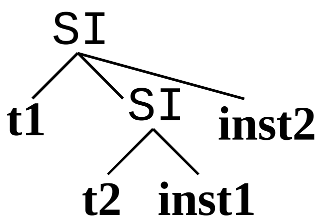


  ✓ **Instruction →** `SI` **test** `ALORS` **Instruction**
  | `SI` **test** `ALORS` **Instruction** `SINON` **Instruction**

  `SI` **t1** `ALORS` `SI` **t2** `ALORS` **inst1** `SINON` **inst2**



ou



  ✓ …

❑ Yacc

- **Fichier d'entrée pour Yacc : (comme pour Lex).**

  fichier dans lequel on met les règles de grammaire pour l'analyse syntaxique et les actions attachées à ces règles.

- **Yacc prend un fichier d'entrée et génère, à partir de ce dernier, un fichier C contenant toutes les données et procédures permettant l'analyse syntaxique.**

```
mon_fichier_dentree.yacc  --yacc-->  y.tab.c
```

- **Format général d'un fichier d'entrée :**

```
        (I) déclarations
%%
        (II) règles d'analyse et actions
%%
        (III) code utilisateur
```

- **Dans la partie *(I)*, tous les textes compris entre les bornes `%{` et `%}` sont textuellement et intégralement copiés à la même place dans le fichier C généré. Ces bornes `%{` et `%}` doivent être <u>absolument</u> en début de ligne.**

- *(I)* **<u>déclarations</u> :**
  - **L'axiome (si omis, c'est le premier non-terminal de la première règle)**
    `%start E`
  - **L'union pour la valeur associée** `yylval`
    ```
    %union { EXPR_ARBRE arbre; char *chaine;
             int entier; char caractere ; }
    ```
  - **Les tokens (pour "typer " les valeurs associées aux tokens)**
    ```
    %token <chaine> VARIABLE AUTRES
    %token <entier> CONSTANTE
    %token <caractere> OPERATEUR
    %token PARENTHESE_OUVRANTE PARENTHESE_FERMANTE
    ```
  - **<u>Chaque caractère ASCII est son propre token (voir ci-dessous)</u>.**
  - **Le type (pour "typer " les valeurs associées aux non-terminaux)**
    `%type <arbre> E T F`
  - **Pour résoudre certains conflits de type shift/reduce concernant les opérateurs, Yacc propose de déclarer leurs précédences avec**
    `%right %left %nonassoc`

Par exemple :

`%left '+' '-'`	*associtivité gauche*
`%left '*' '/'`	*associtivité gauche*
`%nonassoc MOINS_UNAIRE`	*non associatif, donc unaire*

L’ordre des déclarations indique l’ordre de précédence.

Pour notre exemple ci-dessus,

- **les opérateurs `'+'` et `'-'` ont une précédence supérieure aux opérateurs définis avant (s’il y en a),**
- **les opérateurs `'*'` et `'/'` ont une précédence supérieure aux `'+'` et `'-'`,**
- **l’opérateur `'-'` (moins unaire) ont une précédence supérieure aux `'*'` et `'/'`.**

  L’utilisation de `MOINS_UNAIRE` pour l’opérateur `'-'` (moins unaire) sera montré dans un exemple un peu plus loin.

- *(II)* **<u>règles d'analyse et actions</u> :**
  - **On y déclare les règles : (en général sous la forme ci-dessous)**

    règle ***actions_composées_d'instructions_C***

  - **L'utilisation des `$$`, `$1`, .., `$n` pour indiquer les valeurs associées aux symboles d'une règle.**

    Par exemple :

```
E : E '+' T       { $$ = CreerBin('+', $1, $3); }
     | E '-' T    { $$ = CreerBin($2, $1, $3); }
     | T          { $$ = $1; }
;
...
F : '(' E ')'     { $$ = $2; }
     | VARIABLE   { $$ = CreerVariable($1); }
     | CONSTANTE  { $$ = CreerConstante($1); }
;
```

- *(III)* **<u>code utilisateur</u> :**
  - **On y inclut (en général) le fichier généré par Lex pour l'analyse lexicale.**
  - **Yacc, au moment où il en a besoin, appellera la fonction** `yylex()`**.**
  - **Yacc fournit à Lex la variable** `yylval`**.**
  - **L'appel à l'analyseur syntaxique (appelé aussi parseur) est fait par** `yyparse()`**. Cette dernière pourrait avoir une valeur de retour :**
    - `0` **lorsque l'analyseur syntaxique est satisfait et que l'analyseur lexicale envoie le marqueur de fin accepté par l'analyseur syntaxique,**
    - `1` **en cas d'erreur détectées.**
  - **Penser aussi à redéfinir la procédure** `void yyerror (char *s);`**.**
  - **On génère à partir du fichier d'entrée pour Yacc avec**

```
yacc fichier_entree_pour_yacc.yacc
```

❑ Ocamlyacc

- **Fichier d'entrée pour ocamlyacc**

  fichier dans lequel on met les règles de grammaire pour l'analyse syntaxique et les actions attachées à ces règles.

- **ocamlyacc prend un fichier d'entrée et génère, à partir de ce dernier, un fichier ocaml suffixé par `ml` contenant toutes les données et procédures permettant l'analyse syntaxique.**

$$\boxed{\texttt{nom.mly}} \xrightarrow{\texttt{ocamlyacc}} \boxed{\texttt{nom.ml}}$$

- **Format général d'un fichier d'entrée :**

```
            (I) déclarations
%%
            (II) règles d'analyse et actions
%%
            (III) code utilisateur
```

- *(I)* **<u>déclarations</u> :**
  - **tous les textes compris entre les bornes** `%{` **et** `%}` **(à déclarer en premier) sont textuellement et intégralement copiés dans le fichier ml généré.**
  - **Au moins un axiome doit être déclaré (on peut déclarer plusieurs et chacun se transformera en une fonction)**

    ```
    %start E
    ```
  - **Les tokens**

    ```
    %token <int>    TOKEN_ENTIER
    %token <string> TOKEN_VARIABLE
    %token <char>   TOKEN_PLUS_MOINS TOKEN_MULT_DIV
    %token     TOKEN_OUVRANTE TOKEN_FERMANTE TOKEN_FIN
    ```

    Ils permettront de générer le type token.
  - **Typer les valeurs associées aux non-terminaux**

    ```
    %type <Exemple2_type.expr_arbre> eE e t f
    ```
  - **Attention : les non-terminaux commencent par une lettre minuscule.**
  - **Les commentaires :** `/* blablabla */`

- **Pour résoudre certains conflits de type shift/reduce concernant les opérateurs comme en Yacc.**

  `%right %left %nonassoc`

- *(II)* **<u>règles d'analyse et actions</u> :**
  - **Même hose qu'en Yacc**
  - **L'utilisation des** `$1`**, .., ** `$n` **pour indiquer les valeurs associées aux symboles d'une règle.**
  - **Les commentaires :** `/* blablabla */`

- *(III)* **<u>code utilisateur</u> :**
  - **Ne pas faire appel aux fonctions générées à partir des non-terminaux à cause du typage du token. Il vaut mieux créer un fichier indépendant qui appellera les analyseurs syntaxique et lexical (voir Exemple2 ci-dessous).**

❑ Lex et Yacc pour les expressions arithmétiques traditionnelles

- **Fichier d'entrée pour Lex :** `Exemple2.lex`

```
%{
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "types.h"
%}

ER_ENTIER          -?[0-9]+
ER_VARIABLE        [a-zA-Z][0-9a-zA-Z]*
ER_SEPARATEUR      [ \n\t]

%%

{ER_SEPARATEUR}+ {}

{ER_ENTIER} {
  yylval.entier = atoi(yytext);
  return(TOKEN_ENTIER);
}
```

```
{ER_VARIABLE} {
  yylval.chaine = strcpy((char *)malloc(yyleng+1), yytext);
  return(TOKEN_VARIABLE);
}

[\+\-] {                  /* ou [-+] */
  yylval.caractere = yytext[0];
  return(TOKEN_PLUS_MOINS);
}

[\*\/] {                  /* ou [*/] */
  yylval.caractere = yytext[0];
  return(TOKEN_MULT_DIV);
}

[\(\)] {                  /* ou [()] */
  return(yytext[0]);
}

<<EOF>> {
  return(0);              /* pour indiquer a Yacc la fin */
}

%%
```

- **Fichier d'entrée pour Yacc :** `Exemple2.yacc`

```
%{
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "types.h"

EXPR_ARBRE a;
%}

%union { EXPR_ARBRE arbre; char *chaine; int entier; char caractere; }

%token <chaine>           TOKEN_VARIABLE
%token <entier>           TOKEN_ENTIER
%token <caractere>        TOKEN_PLUS_MOINS  TOKEN_MULT_DIV
%token                    TOKEN_FIN

%type <arbre>    EE  E  T  F

%start EE

%%

EE :  E     { a = $1; }

;
```

```
E :  E  TOKEN_PLUS_MOINS  T       { $$ = CreerBin($2, $1, $3); }
  |  T                            { /* par defaut, $$ = $1 */  }
;
T :  T  TOKEN_MULT_DIV  F         { $$ = CreerBin($2, $1, $3); }
  |  F                            { }
;
F :  '('  E  ')'                  { $$ = $2; }
  |  TOKEN_VARIABLE               { $$ = CreerVariable($1); }
;
F :  TOKEN_ENTIER                 { $$ = CreerConstante($1); }
;
%%
```

```
#include "lex.yy.c"


int
main (int argc, char **argv)
{
  yyparse();

  ImprimerPrefixe(a);
  printf("\n\n");
  ImprimerPostfixe(a);
  printf("\n\n");
  ImprimerParenthese(a);
  printf("\n\n");

  return 0;
}


void
yyerror(char *s)
{
  fprintf( stderr, "%s\n", s );
}
```

- **Un exemple avec des règles ambiguës provoquant des conflits shift/reduce qui sont résolus par l'utilisation des `%right %left %nonassoc`.**

  (Penser à modifier dans le fichier d'entrée pour Lex les règles concernant les `'+'`, `'-'`, `'*'`, `'/'`, `'('` et `')'`).

```
...
%left '+' '-'
%left '*' '/'
%nonassoc MOINS_UNAIRE
...
%%
...
E : E '+' E                         { $$ = CreerBin('+', $1, $3); }
  | E '-' E                         { $$ = CreerBin('-', $1, $3); }
  | E '*' E                         { $$ = CreerBin('*', $1, $3); }
  | E '/' E                         { $$ = CreerBin('/', $1, $3); }
  | '-' E  %prec  MOINS_UNAIRE          { $$ = CreerUn('-', $2); }
  | '(' E ')'                       { $$ = $2; }
  | ...
;
...
%%
...
```

- **Quelques remarques concernant le fichier d'entrée pour Yacc :**

  - **Grammaire non-contextuelle,**
  - **On déclare un non-terminal, ici** `EE` **comme axiome par** `%start EE`**. Par défaut, Yacc choisira le premier non-terminal de la première règle d'analyse.**
  - **La grammaire originale est augmentée avec la règle**

    `EE → E  <<EOF>>`
  - **Chaque règle d'analyse commence par une règle de grammaire (: à la place de →) suivie d'une action (instruction C) :**

    ```
    E : E '+' T   { code C };
    A :           { code C };
    ```
  - **Chaque règle doit retourner une valeur indiquée souvent dans l'action par l'instruction** `$$ =` **expression.**
  - **Chaque règle d'analyse est indépendante des autres. Pour faciliter la communication des valeurs de retour entre ces règles, on utilise le pseudo-variable** `$1` **(resp.** `$2`**, etc, …) pour accéder à la valeur retournée par le 1er (resp. 2ème, etc, …) membre à droite du " : ".**
  - **Les valeurs retournées sont de type varié. Elles sont donc regroupées dans une structure d'union** `%union`**.**

- **Par défaut, on a** `$$ = $1`**.**
- **Pour pouvoir utiliser la valeur de retour d'un non-terminal, ce dernier doit être "typé" (** `%type <arbre> EE  E  T  F`**) et Yacc s'occupe du reste grâce aux** `$$ =` **expression.**
- **Pour pouvoir utiliser la valeur de retour d'un terminal (token), ce dernier doit être "typé" (** `%token <chaine> TOKEN_VARIABLE`**) et Lex s'occupe du reste grâce à la variable** `yylval`

```
yylval.chaine = strcpy((char *)malloc(yyleng+1),
                                        yytext);
```

- **Attention, on parle de "typer" au sens Yacc : on "type" avec un champ de la structure d'union.**

- **L'ordre des règles n'a aucune importance.**
- **On utilise le symbole "** `|` **" pour séparer les règles d'un même non-terminal :**

```
F :  '('  E  ')'    { $$ = $2; }
  |  TOKEN_VARIABLE { $$ = CreerVariable($1); }
;
```

- **Chaque règle ou groupe de règles se termine avec "** `;` **".**

– **On appelle l'analyseur syntaxique par la fonction** `yyparse()`**. S'il y a une erreur, cette dernière retournera la valeur** `1`**.**

Si l'analyseur lexical retourne la valeur de marqueur de fin (ici 0 par `<<EOF>> { return(0); }` **) et que l'analyseur syntaxique l'accepte et n'attend plus rien (par** `EE : E { a = $1; }` **), alors** `yyparse()` **retournera** `0`**.**

– **L'utilisateur doit définir la procédure** `void yyerror(char *s);` **qui sera appelée lors des erreurs.**

❑ Ocamllex et Ocamlyacc pour les expressions arithmétiques traditionnelles

- **Fichier d'entrée pour ocamllex :** `Exemple2_ocamllex.mll`

```
{                       (* (I) *)
open Exemple2_ocamlyacc;;
open Lexing;;
open String;;
}

                        (* (II) *)

let ER_ENTIER           = '-'?['0'-'9']+
let ER_VARIABLE         = ['a'-'z' 'A'-'Z']
                             ['a'-'z' 'A'-'Z' '0'-'9']*
let ER_SEPARATEUR       = [' ' '\n' '\t']

rule analyseur_lexical = parse
     ER_SEPARATEUR+     { analyseur_lexical lexbuf }
   | (ER_ENTIER as x)   { TOKEN_ENTIER (int_of_string x) }
   | (ER_VARIABLE as x)        { TOKEN_VARIABLE x }
   | (['+' '-'] as x)   { TOKEN_PLUS_MOINS x }
```

```
    | (['*' '/'] as x)  { TOKEN_MULT_DIV x }
    | '('               { TOKEN_OUVRANTE }
    | ')'               { TOKEN_FERMANTE }
    | eof               { TOKEN_FIN }


{                       (* (III) *)
}
```

- **Un fichier des types utilisés par l'analyseur syntaxique :** `Exemple2_type.mli`

```
type expr_arbre =
    Constante of int
  | Variable of string
  | Binaire of char * expr_arbre * expr_arbre
;;
```

- **Fichier d'entrée pour ocamlyacc :** `Exemple2_ocamlyacc.mly`

```
%{
open Exemple2_type;;
%}

%token <int>   TOKEN_ENTIER
%token <string> TOKEN_VARIABLE
%token <char>  TOKEN_PLUS_MOINS TOKEN_MULT_DIV
%token         TOKEN_OUVRANTE TOKEN_FERMANTE TOKEN_FIN

%type <Exemple2_type.expr_arbre> eE e t f

%start eE

%%

eE : e TOKEN_FIN      { $1 }
;

e : e TOKEN_PLUS_MOINS t      { Binaire ($2, $1, $3) }
  | t                         { $1 }
;
```

```
t : t TOKEN_MULT_DIV f        { Binaire ($2, $1, $3) }
  | f                         { $1 }
;

f : TOKEN_OUVRANTE  e TOKEN_FERMANTE { $2 }
  | TOKEN_VARIABLE                   { Variable $1 }
  | TOKEN_ENTIER                     { Constante $1 }
;

%%

(* ATTENTION : Dans cette partie, les commentaires sont
       OCaml *)
(*
  ATTENTION : ne pas utiliser ici l'axiome a cause d'un
  probleme de typage entre le Exemple2_ocamlyacc.token
  du lexer et le token du parseur.
  Il vaut mieux le faire avec un fichier exterieur (ici
  Exemple2_ocaml.ml qui fait un open de
  Exemple2_ocamlyacc. Plus de problème de typage.
*)
```

- **Fichier utilisant les analyseurs lexical et syntaxique :** `Exemple2_ocaml.ml`

```
open Exemple2_type;;
open Exemple2_ocamlyacc;;
open Printf;;

let rec imprimer_prefixe a =
  match a with
      Constante x       -> printf "%d" x
    | Variable x        -> printf "%s" x
    | Binaire (x, y, z) -> printf "%c " x;
                           imprimer_prefixe y;
                           printf " ";
                           imprimer_prefixe z;
;;

let lexbuf = Lexing.from_channel stdin in
  let resultat =
        eE Exemple2_ocamllex.analyseur_lexical lexbuf in
    imprimer_prefixe resultat;
    printf "\n";
    flush stdout
;;
```

Bibliographie

[1] Alfred Aho, Ravi Sethi & Jeffrey Ullman
Compilateurs : principes, techniques et outils
Paris, Dunod (La référence et toujours d'actualité)

[2] D. Beauquier, J. Berstel & Ph. Chrétienne
Eléments d'algorithmique
Masson

e-Bibliographie

[1e] http://pages.cs.wisc.edu/~larus/spim.html

[2e] http://dinosaur.compilertools.net/lex
http://flex.sourceforge.net

[3e] *http://www.linux-kheops.com/doc/ansi-c*

[4e] http://logos.cs.uic.edu/366/notes/MIPS%20Quick%20Tutorial.htm

[5e] http://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2005/ue/arcmo-2005oct/_fichiers/Li321_lang_asm.pdf

[6e] http://fr.wikipedia.org/wiki/Architecture_MIPS