Université de Nice - Sophia Antipolis

Licence 3 Informatique

# Techniques de compilation

## Introduction à Lex/Flex et Bison/YACC

Jacques Farré

e-mail : Jacques.Farre@unice.fr

# Table des matières

# 1 Le couple Lex-YACC

On présente ici les principales caractéristiques de Lex/Flex et YACC/Bison. Malheureusement, il y a des différences parfois notables entre les différents Lex et Flex. Heureusement, il est possible d'utiliser Flex en mode Lex, et les différences se réduisent, mais il y en a encore.

Les différences entre Bison et YACC sont plus faibles. Bison dispose aussi d'une option pour le faire fonctionner en mode YACC.

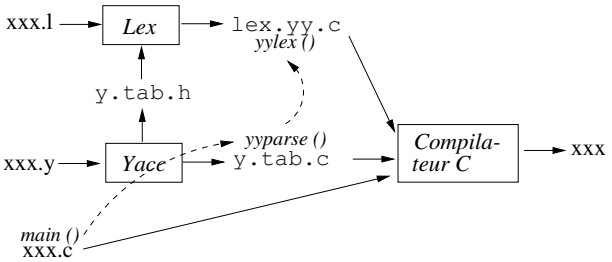
Ces différences seront indiquées autant que faire se peut.

Le nom Lex sera utilisé de manière générique pour les Lex et Flex. De même, le nom YACC sera utilisé de manière générique pour YACC et Bison.

Lex et YACC ne forment pas un couple inséparable. On peut utiliser Lex sans YACC, si on a uniquement besoin de reconnaître des expressions régulières.

On peut aussi utiliser YACC sans Lex, ce qui est souvent le cas. Il suffit de savoir que YACC, pour avancer sur les lexèmes, appelle une fonction `int yylex ()`. Cette fonction doit renvoyer le code du caractère pour les lexèmes formés d'un seul caractère, et le code défini dans `y.tab.h` (`nom.tab.h` pour `nom.y` avec Bison) pour ceux déclarés à l'aide d'un identificateur dans YACC (`%token`). Enfin, cette fonction doit renvoyer 0 pour la fin de fichier.

## Le couple Lex-YACC




- `lex xxx.l` produit un programme C dans `lex.yy.c`
  `yacc xxx.y` produit un programme C dans `y.tab.c` (option `-d` ⇒ définitions dans `y.tab.h`), `bison xxx.y` ⇒ `xxx.tab.c`, `xxx.tab.h`
- compiler, relier avec les bibliothèques :
  `cc lex.yy.c y.tab.c xxx.c -ll -ly` (`-lfl` à la place `-ll` pour Flex)

**Slide 1**

# 2 Lex

Lex a été crée en 1975 par A. E. Lesk, des laboratoires Bell.

La version de GNU est Flex. Il existe une version de FLex pour Ada (Aflex), et une pour Eiffel (Flex/Eiffel). Quant à C++, une version de Lex est disponible avec YACC++.

Flex ajoute des fonctionnalités à Lex (analyse sur plusieurs fichiers, analyse à partir de chaînes en mémoire ...) qui ne seront pas exposées ici.

Une version acceptant des caractères sur 16 bits existe aussi, Zflex.

## Lex : un générateur d'analyseurs lexicaux

- repose sur les expressions régulières
- produit un automate à état finis
- forme générale d'un texte Lex

```
[definitions]
%%
[regles]
[%%
sous-programmes
]
```

  sauf le premier `%%`, tout ce qui est indiqué entre [ ] est optionnel.
- les messages d'erreur de Lex ne sont pas souvent très explicites, `syntax error`, et sans n$^o$ de ligne !

**Slide 2**

## 2.1 Caractères spéciaux de Lex

Ajouter aux tableaux qui suivent, pour Flex, la forme `[:xxx:]`, par exemple `[:alnum:]`, qui reconnaît les mêmes caractères que la fonction de la bibliothèque standard de C `isXXX`, par exemple `isalnum`.

### Caractères spéciaux de Lex

- Tous les caractères sont significatifs, y compris les espaces.
- Les caractères suivants sont réservés : ils doivent être notés `\x` ou `"x"` si l'on veut leur valeur littérale

| caractère | signification | exemple | produit/reconnaît |
|---|---|---|---|
| `+` | 1 ou plusieurs fois | `x+` | `x...x` |
| `*` | 0 ou plusieurs fois | `t*` | vide ou `t...t` |
| `?` | 0 ou 1 fois | `a?` | vide ou `a` |
| `\|` | union | `ab\|bc` | `ab` ou `bc` |
| `(...)` | factorisation | `(a\|b)c` | `ac` ou `bc` |

- les priorités sont, en ordre décroissant : `+`, `*`, `?`, puis concaténation (`xyz`), puis `|`.

**Slide 3**

## Autres caractères spéciaux de Lex

| carac. | signification | exemple | produit/reconnaît |
|---|---|---|---|
| " | valeur littérale des car. | "+?"+ | +?...+? |
| \ | idem (avec convention C) | \+\?+ | +?...? |
| | \ spécial même entre " | "\"+ | erreur |
| | \NNN ⇒ code octal | \040 | un espace |
| | \xHH ⇒ code hexa | \xA0 | un espace |
| . | tout car. sauf fin de ligne | .\|\n | tous les caractères |
| [...] | ensemble de caractères | [aeiou] | les voyelles |
| - | intervalle d'ensemble | [0-9] | chiffres |
| ^ | complément d'ensemble | [^0-9] | tout car. sauf chiffre |

Attention. [...] = ensemble de caractères, pas d'expressions régulières :
[(0|1)+] = un des caractères (, ), 0, 1, |, +, pas une suite de 0 ou 1.

**Slide 4**

## Autres caractères spéciaux de Lex - suite

| carac. | signification | exemple |
|---|---|---|
| { et } | nom d'une expression régulière | {ID} |
| | ou une répétition | x{1,5} |
| | | x{1,} |
| | | x{3} |
| %% | sépare les sections du source Lex | |
| <<EOF>> | fin de fichier | |
| / | reconnaissance dans un contexte droit | x/y |
| $ | reconnaissance en fin de ligne | " "$ |
| ^ | reconnaissance en début de ligne | #^ |
| <...> | reconnaissance conditionnelle | <OCT>[0-7] |

Ce dernier groupe peut réserver des surprises quand on les combine

**Slide 5**

## 2.2 Définitions

### Définitions d'expressions régulières

Les expressions ci-contre reconnaissent des identificateurs comme

```
a    PI      n23    r2d2   pointeur_de_pile_3
```

mais pas comme

```
a_   _pi    n__2
```

Attention : Lex procède par substitution textuelle. Si dans l'exemple ci-contre, la définition de `LouC` ne comprenait pas de `( )`, l'expression serait équivalente à :

```
[a-zA-Z](_?[a-zA-Z]|[0-9])*
```

c'est à dire que le _ ne pourrait pas précéder un chiffre.

### Définitions en C

Tout ce qui est noté entre `%{` et `%}` (en début de ligne) dans les définitions est considéré comme du texte C, et sera inséré tel quel dans le fichier `lex.yy.c`. Par exemple :

```
%{
#define MAX 100

static int compteur;

extern int Hash (const char*);
%}
```

Définitions d'expressions régulières

- La section de définitions sert à nommer des expressions régulières
- L'identificateur doit être séparé de l'expression par au moins un espace
- Par exemple, l'expression régulière d'un identificateur qui doit commencer par une lettre, éventuellement suivie de lettres ou de chiffres pouvant être séparés par un _ s'écrit :

```
[a-zA-Z](_?[a-zA-Z0-9])*
```

- On peut rendre son écriture plus claire

```
Lettre   [a-zA-Z]
Chiffre  [0-9]
LouC     ({Lettre}|{Chiffre})
%%
{Lettre}(_?{LouC})*
```

- C'est une substitution textuelle

**Slide 6**

## 2.3 Règles

- Les règles sont composées d'expressions régulières et éventuellement d'actions.
- Les expressions régulières doivent être séparées des actions par au moins un espace (blanc ou tabulation).
- Les actions sont des instructions C. S'il y a plusieurs actions, elles doivent être entre { et }, et en ce cas elles peuvent tenir sur plusieurs lignes.
- ne pas indiquer d'action revient à définir une action vide.
- l'action notée `|` indique qu'il faut faire l'action de la ligne suivante

```
bleu    |
jaune   |
rouge   |
vert     printf ("couleur\n");
```

- si une chaîne n'est pas reconnue, elle est écrite sur *output*

Une fois une chaîne reconnue, l'action correspondante est effectuée. Une nouvelle chaîne de la suite du texte peut alors être reconnue.

La chaîne reconnue est rangée dans un tableau de caractères nommé `yytext` (attention, `char yytext [] pour Lex`, et `char* yytext` pour Flex par défaut), et dont la longueur effective est rangée dans la variable `yyleng` (de type `int`).

### Règles

- de la forme

```
expression-reguliere action
```

- par exemple

```
%{
static int nb_mots = 0;
%}
%%
[+*/-]      printf ("operateur\n");
[a-z]+      |
[A-Z]+      { printf ("mot %s\n", yytext); ++nb_mot; }
" "+        printf ("%d espaces consesutifs\n", yyleng);
[ \t]       ;
```

**Slide 7**

Lex analyse le texte source, et s'arrête sur la plus longue chaîne qui peut être engendrée par une des expressions régulières.

Si plusieurs expressions régulières peuvent reconnaître une même chaîne, c'est la première qui est retenue.

Par exemple, ci-contre, le mot `begin` est engendré par les deux expressions régulières, c'est la première qui est retenue. Dans le mot `beginning`, `begin` répond à la première expression, mais la seconde permet de reconnaître une phrase plus longue et est retenue.

Cette règle est bien pratique, mais peut causer des surprises. Par exemple, l'expression régulière :

```
\".*\"
```

ressemble à la définition d'une chaîne de caractères (n'importe quel caractère - sauf fin de ligne - entre "). Mais dans le texte :

```
"Lex" + "nous joue" + "des tours"
```

Lex ne reconnaîtra qu'une seule chaîne au lieu de trois, et c'est normal vu la définition donnée.

Il faut aussi dire qu'une chaîne ne peut contenir de " (sauf par exemple s'il est doublé). La bonne expression est donc :

```
\"([^"\n]|\"\")*\"
```

Enfin, Lex ajoute toujours implicitement la règle:

```
.|\n  ECHO
```

qui affiche donc toutes les portions du texte qui lui est soumis et qui ne correspondent à aucune des expressions régulières données. Donc, ajoutez comme dernière règle

```
.|\n   ;
```

si vous voulez qu'il n'y ait pas d'affichage de se qui n'est pas reconnu.

## Choix des règles et des actions

- par exemple

```
%%
begin    printf ("mot cle\n");
[a-z]+   printf ("ident\n");
```

  - `begin` ⇒ mot clé
  - `beginning` ⇒ ident

- Attention :

```
\".*\"
```

  incorrect pour reconnaître deux chaînes dans une expression comme `"abcde" + "xyz"` : les deuxième et troisième guillements sont absorbés par `.*`
- forme correcte :

```
\"([^"\n]|\"\")*\"
```

- règle implicitement ajoutée par Lex

```
.|\n    ECHO
```

**Slide 8**

## 2.4 Fonctions et macros

La dernière partie d'un source Lex peut contenir des fonctions C, qui peuvent bien entendu être appelées dans les actions.

Lex utilise un certains nombre de fonctions et macros prédéfinies. Le texte des macros peut être vu dans le fichier C produit par Lex (`lex.yy.c`).

### yywrap

La fonction `yywrap ()` est appelée quand la fin du texte analysé est rencontrée. Lex s'arrête lorsque `yywrap` renvoie 1, sinon il continue de lire (si bien que si vous ne changez pas de fichier d'entrée, vous passerez votre temps à appeler `yywrap`). Il y a un `yywrap` par défaut, qui se contente de renvoyer 1. Pour l'obtenir, il faut lier avec la bibliothèque Lex : `-ll` (Flex : `-lfl`).

On redéfinit `yywrap` quand on a besoin d'effectuer des actions après avoir traité le texte, comme dans l'exemple ci-contre.

La redéfinition de la fonction `yywrap ()` peut par exemple être utilisée pour changer de fichier d'entrée une fois que le fichier courant a été analysé (traitement des `#include` de C par exemple).

### ECHO

La macro `ECHO` est équivalente à `printf ("%s", yytext)`.

### output

`output (c)` écrit le caractère `c` sur `yyout`, qui est assigné par défaut à `stdout`. On peut évidemment lui assigner un autre fichier.

Fonction `yywrap ()`

La commande `wc` de Unix en Lex

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

static int car = 0, lig = 0, mot = 0;
%}

%%

\n          { ++car; ++lig; }
[^ \t\n]+   { car += yyleng; ++mot; }
.           ++car;

%%

int yywrap () {
  printf ("%7d %7d %7d\n", lig, mot, car);
  return 1;
}
```

**Slide 9**

### input

`input ()` lit le prochain caractère du flot d'entrée. La lecture se fait sur `yyin` (de type `FILE*`), qui est assignée par défaut à `stdin`. On peut lire sur une autre entrée simplement en assignant un autre fichier à `yyin`.

La redéfinition de `input` peut permettre de simplifier les expressions régulières. Par exemple si le langage à analyser ne fait aucune différence entre lettres minuscules et majuscules, elle peut transformer toutes les majuscules en minuscules. Cela évite d'écrire des définitions fastidieuses du genre :

```
a           [aA]
...
z           [zZ]
%%
{w}{h}{i}{l}{e}    return WHILE;
 ...
```

Jetez un coup d'oeil à la macro `input` dans un fichier `lex.yy.c`, et essayez de comprendre ce qu'elle fait ! Essayez de la transformer pour que toutes les lettres majuscules soient transformées en minuscules. La redéfinition se fera dans le source Lex, dans la partie définition :

```
%{
#undef input
#define input ...
%}
LETTRE  [a-z]
```

### unput

`unput (c)` remet le caractère `c` dans le flot d'entrée (`c` sera donc le prochain caractère lu). Peut être répétée. **Attention, avec Flex, ne l'utiliser que si `yytext` est un tableau, pas un pointeur sur caractères** (mettre `%array` dans les définitions).

## Autres fonctions et macros utiles de Lex

- `input ()` : chargée de la lecture des caractères
- `output (char c)` : écrit le caractère `c` sur output
- `unput (char c)` : remet le caractère `c` dans le flot d'entrée
- `yyless (int n)` : permet de revenir `yyleng - n` caractères en arrière
- `yymore ()` : permet de concatèner ce qui a été reconnu avec ce qui
- `yyrestart (FILE* f)` : recommencer l'analyse avec un autre fichier
- `REJECT` : permet de rejetter l'action à une autre expression régulière

**Slide 10**

**yyless**

`yyless(n)` permet de revenir de `yyleng - n` caractères en arrière sur le texte source, c'est à dire de faire comme si seuls les n premiers caractères de la chaîne avaient été reconnus et de remettre les autres dans le flot d'entrée :

```
C [0-9]
%%
".."            printf ("..\n");
{C}+".."        {yyless (yyleng-2); printf ("nombre\n");}
{C}+("."{C}*)?  printf ("nombre\n");}
.|\n            ;
```

Dans le langage reconnu par les expressions régulières de cet exemple, on suppose que les nombres réels peuvent se terminer par un point, sans chiffres pour la partie décimale. Mais deux entiers peuvent être utilisés pour construire un intervalle : `1..10`; ce serait une erreur dans ce cas de considérer que `1.` forme un nombre réel, ce qui se produirait s'il n'y avait pas la ligne contenant `yyless`. Cette ligne permet de "capturer" les nombres suivis par `..` et de remettre les `..` dans le flot d'entrée pour être reconnus ultérieurement.

`yyless (0)` a pour résultat de faire boucler l'analyseur, puisque qu'il n'avance pas sur le texte.

**yymore**

`yymore ()` permet de concaténer les textes successivement reconnus par plusieurs expressions régulières. Voici une façon de reconnaître des chaînes de caractères où un `"` au milieu de la chaîne doit être écrit `\"` :

```
\"[^"\n]*\" {
     if (yytext[yyleng-2] == '\\') {
        /* termine par \" ==> ce n'est pas le " final */
        yyless(yyleng-1); /* remettre " dans le flot d'entree */
        yymore();         /* conserver ce qui a ete reconnu */
     } else printf ("la chaine totale est %s\n", yytext);
 }
```

La chaîne `"abc\"def"` sera reconnue comme s'il y avait concaténation des deux sous chaînes `"abc\` et `"def"`, avec comme valeur finale de `abc"def`.

**REJECT**

La macro `REJECT` sert à faire reconnaître des portions d'une même partie de texte par plusieurs expressions régulières. Voici une petite définition Lex permettant d'afficher toutes les sous-suites croissantes d'une suite de chiffres :

```
%%
[0-9]+                  REJECT;
0*1*2*3*4*5*6*7*8*9*    printf ("%s\n", yytext);
.\n                     ;
```

Le fonctionnement est le suivant. Une suite quelconque (c'est à dire non ordonnée) de chiffres est reconnue car c'est la plus longue chaîne reconnaissable par `[0-9]+`. Comme cette suite est rejetée (macro `REJECT`), on va essayer les autres possibilités de reconnaissance (ici, une suite croissante). La plus longue est encore retenue, car les règles de résolution des ambiguïtés continuent de s'appliquer. Et ainsi de suite, en diminuant la portion de texte tant qu'aucune chaîne n'est reconnue, ou que les chaînes reconnues sont rejetées.

Par exemple, sur la suite `1232123`, les chaînes retenues sont successivement :

| | |
|---|---|
| 1232123 | reconnue et rejetée par la 1ère action, pas reconnue par la 2ème |
| 123212 | idem |
| 12321 | idem |
| 1232 | idem |
| 123 | reconnue et rejetée par la 1ère action, reconnue par la 2ème; on passe donc à la suite du texte |
| 2123 | reconnue et rejetée par la 1ère action, pas reconnue par la 2ème |
| 212 | idem |
| 21 | idem |
| 2 | reconnue et rejetée par la 1ère action, reconnue par la 2ème |
| 123 | reconnue et rejetée par la 1ère action, reconnue par la 2ème |

Notez que si la deuxième action contenait un `REJECT`, on aurait toutes les sous-suites croissantes, et pas seulement les plus longues (123 → 123, 12, 1 puis 23 → 23, 2, puis 3→ 3). Cet exemple montre que l'emploi de `REJECT` est parfois très délicat.

L'emploi de `REJECT` ralentit considérablement l'analyseur.

## 2.5 Contextes

On peut vouloir qu'une chaîne soit reconnue uniquement dans certains contextes.

### Contexte droit

La forme `er1/er2` permet de reconnaître le texte produit par l'expression régulière `er1` seulement si elle est suivie par une chaîne produite par l'expression régulière `er2`. Par exemple

```
C [0-9]
%%
".."              printf ("..\n");
{C}+("."{C}+)?    printf ("nombre\n");
{C}+"."/[^.]      printf ("nombre\n");
.|\n              ;
```

est une autre façon de reconnaître correctement `1..10`.

Attention au piège : le texte correspondant à l'expression régulière après le `/` compte pour la longueur de l'expression régulière reconnue. Si bien que, si les deux lignes reconnaissant un nombre étaient inversées dans l'exemple précédent, `1.3` serait analysé comme `1.`, puis `3`. En effet, `1.3` est de la forme `C+.` non suivie d'un `.` et aussi de la forme `C+ . C+`. Puisque le texte reconnu est de même longueur sur ce cas, c'est la première des deux expressions régulières qui serait retenue.

### Contextes

- `/` pour le contexte droit : problème Fortran

```
DO 5 I = 1, 10       boucle do
DO 5 I = 1. 10       DO5I = 1.10
```

  Solution pour reconnaître le mot clé `DO`

```
DO/[A-Z0-9]+=[A-Z0-9]+,
```

- start conditions pour le contexte gauche :
  `010` = nombre octal, valeur 8
  `10` = nombre décimal, valeur 10
  il faut se rappeler que le 1er chiffre est un 0

```
%START OCT DEC
%{
static int v;
#define VAL (yytext [0] - '0')
%}
%%
<INITIAL>0     { v = 0; BEGIN (OCT); }
<INITIAL>[1-9] { v = VAL; BEGIN (DEC); }
<OCT>[0-7]     v = v * 8 + VAL;
<DEC>[0-9]     v = v * 10 + VAL;
.              { BEGIN (0); }
```

**Slide 11**

**Contexte gauche**

Il permet de reconnaître une chaîne en fonction du texte qui la précède. Ceci est fait à l'aide de "start conditions". Les starts conditions doivent être déclarées par `%START` dans la partie définitions.

Dans Flex, écrire `%s` ou `%x`.

Une start condition peut être indiquée en tête d'une expression régulière, écrite entre < et >. En ce cas, l'expression régulière n'est "retenue" que si l'analyseur est positionné dans cette condition.

On positionne l'analyseur dans une condition par la macro `BEGIN (condition)`.

`BEGIN (0)` positionne l'analyseur en condition initiale, nommée comme il se doit `INITIAL` (on peut donc aussi écrire `BEGIN (INITIAL)`). Comme les start conditions sont en fait des valeurs entières, il est possible d'écrire `BEGIN (i)`, si `i` a une valeur de start condition. Il est possible d'obtenir la valeur de la start condition courante par la macro `YYSTATE`: `i = YYSTATE`. En Flex, cette macro est appelée `YY_START`.

Il est possible de donner plusieurs contextes : `<C1,C2,...,Cn>exp.reg`.

Une expression régulière qui n'est pas précédée par une start condition est reconnue dans tous les contextes. C'est en fait un peu plus compliqué que cela avec Flex, qui distingue les conditions inclusives (`%s`), identiques à celles de Lex, et les conditions exclusives (`%x`), pour lesquelles une expression régulière ne sera reconnue dans tous les contextes que si elle esr précédée de `<*>`.

L'exemple suivant reconnaît des nombres écrits en décimal (`123` par exemple), en octal (`O777` par exemple) ou en hexadécimal (`HFFFF` par exemple). Les nombres doivent être séparés par au moins un espace (blanc, fin de ligne ou tabulation). On affiche leur valeur, et on traite les erreurs (caractère qui n'est pas un chiffre ou un espace, chiffre qui n'appartient pas à la base, ...), en utilisant le minimum possible de code C.

```
%START DECIMAL OCTAL HEXA ERREUR

Oct   [01234567]
Dec   {Oct}|[89]
hex   [abcdef]
HEX   [ABCDEF]
SP    [ \t\n]
XX    [^ \t\n]

%{
int val;
%}

%%
<OCTAL>{Oct}     val = val * 8 + (yytext [0] - '0');
<DECIMAL>{Dec}   val = val * 10 + (yytext [0] - '0');
<HEXA>{Dec}      val = val * 16 + (yytext [0] - '0');
<HEXA>{hex}      val = val * 16 + (yytext [0] - 'a' + 10);
<HEXA>{HEX}      val = val * 16 + (yytext [0] - 'A' + 10);

<INITIAL>{Dec}   {val = yytext [0] - '0'; BEGIN DECIMAL;}
<INITIAL>[oO]    {val = 0; BEGIN OCTAL;}
<INITIAL>[hH]    {val = 0; BEGIN HEXA;}

<ERREUR>{XX}     /* ne pas multiplier les messages d'erreur */;
{XX}             {printf("erreur sur %s\n", yytext); BEGIN ERREUR;}

<ERREUR>{SP}     BEGIN 0;
<INITIAL>{SP}    /* sauter les espaces non significatifs */;
{SP}             {printf ("%d\n", val); BEGIN 0;}
```

## 2.6 Modification des tailles des tables internes de Lex

Ceci se fait dans la partie définition par :

```
%x n
```

où `n` est un entier et `x` un des caractères parmi `[pneako]`. Si vous "explosez" Lex, il vous indique quelle table est saturée, sa taille actuelle, et le caractère `x` associé. Il suffit de définir un n supérieur à la taille donnée dans le message.

## 2.7 Différences entre Lex et Flex

Sauf si l'option `-l` (compatibilité avec Lex) est donnée à la commande Flex, on a les différences suivantes

- la variable `int yylineno`, n° de la ligne courante, n'est pas fournie
- `input` est une fonction en Flex, pas une macro comme en Lex, et ne peut donc pas être redéfinie; mais Flex fourni une macro `YY_INPUT` pour contrôler la lecture du texte d'entrée
- pas de macro `output`
- différence de priorité pour les opérateurs de répétition { et }, et l'opérateur de contexte ^.
- les directives de modification de taille des tables ne sont pas traitées par Flex

#### Fonctions et macros spécifiques à Flex

Ce sont des fonctionnalités assez avancées, qui débordent du cadre de ce cours. Reportez vous au manuel de Flex par Vern Paxson notamment.

- possibilité de produire un analyseur qui est une classe C++
- possibilité de produire des analyseurs interractifs ou non interactifs
- possibilité de lire le texte à partir de chaînes plutôt quà partir de fichiers
- possibilité de définir des actions qui seront effectuées dans tous les cas où une phrase est reconnue
- emboîtement de start conditions

# 3 YACC

Créé en 1974 par S. C. Johnson.

La version de GNU est Bison. Il existe des versions pour Ada (Ayacc), une pour Eiffel (Bison/Eiffel). La version C++, avec certaines extensions, est YACC++. Zyacc et BTyacc offrent quant à eux des extensions intéressantes.

Yacc est invoqué par la commande `yacc options fichier`.

L'option -v produit les tables d'analyse sur le fichier y.output.

L'option -d produit un fichier y.tab.h (`nom.tab.h` avec Bison, si le source est `nom.y`) qui contient les définitions des lexèmes et le type des attributs, pour permettre l'interface avec l'analyseur lexical entre autre.

L'analyseur lexical n'est pas obligatoirement Lex. Il est fréquent qu'il soit programmé à la main, et inclus dans le fichier YACC

```
%{
static int yylex ();
...
%}
%%
...  /* regles */
%%
int yylex () {
  ...
}
```

## YACC : un générateur d'analyseurs syntaxiques

- YACC crée un analyseur LALR(1), auquel il est possible d'ajouter des actions sémantiques et d'indiquer comment réparer les erreurs
- Forme générale d'un texte YACC

```
[definitions]
%%
[regles]
[%%
sous-programmes
]
```

  sauf le premier `%%`, tout ce qui est indiqué entre [ ] est optionnel.
- Les espaces (blanc, fin de ligne, tabulation) ne sont pas significatifs. Les commentaires sont comme en C.

**Slide 12**

## 3.1 Définitions

Les définitions contiennent soit du texte C (entre `%{` et `%}`), soit les déclarations des lexèmes, de l'axiome, soit les types des attributs et des non terminaux. Elles peuvent être répétées et apparaître dans n'importe quel ordre.

**Déclaration de lexème**

En général, dans un langage, les identificateurs, les nombres, les mots-clés se sont pas constitués d'un seul caractère, et sont définis par une expression régulière (cette définition est faite en dehors de YACC). Mais il faut les déclarer comme terminaux à YACC : `%token IDENT NOMBRE KWHILE` Il est inutile de déclarer (mais il n'est pas interdit de le faire) les lexèmes simples (`%token '+'` par exemple).

**Déclaration de lexème avec associativité et priorité (précédence)**

Les déclarations de précédence peuvent servir à résoudre des conflits; elles sont surtout utilisées pour donner des grammaires ambiguës (voir plus loin).

Les déclarations de précédence sont :

- `%left` : associativité à gauche
- `%right` : associativité à droite
- `%nonassoc` : non associatif

La priorité est fonction de l'ordre de déclaration, les moins prioritaires en premier (`*` est plus prioritaire que `+` sur l'exemple ci-contre).

Un lexème défini par `%nonassoc`, `%left`, `%right`, n'a pas besoin d'être défini par `%token`.

Une précédence peut être localement modifiée par `%prec`. Dans l'exemple ci-contre, `-a * b` sera traité comme `(-a) * b` puisque **ce** – est associatif à gauche et est aussi prioritaire que `*`.

### Déclaration des terminaux

- déclaration normale

```
%token IDENT NOMBRE KWHILE
```

- déclaration avec priorité et associativité (pour les grammaires ambiguës)

```
%nonassoc '=' '<' '>' NEQ LEQ GEQ
%left '+' '-'
%left '*' '/'
%right '^'
```

- modification locale de la priorité dans une règle

```
exp   : exp '*' exp
      | '-' exp  %prec '*'
      | ...
```

**Slide 13**

Notez qu'il est possible de déclarer un token sans que celui ci soit utilisé dans les règles de la grammaire. Cette possibilité est utilisée en particulier pour modifier localement une précédence, par exemple

```
%left '*' '/'
%nonassoc MOINS_UNAIRE
%right '^'
```

et

```
exp   : ...
      | '-' exp  %prec MOINS_UNAIRE
```

**Déclaration de l'axiome**

Elle est optionnelle

```
%start toto
```

Par défaut, l'axiome est la partie gauche de la première règle.

**Définition des types des attributs**

Les attributs sont en général une union de types

Le nom de ce type dans le C produit est `YYSTYPE`. La définition de ce type est copiée dans `y.tab.h` (si option `-d`), donc l'analyseur lexical peut gérer les attributs des lexèmes en incluant ce fichier.

Par défaut (c'est à dire en l'absence de `%union`) le type des attributs est entier.

**Définition des attributs des symboles**

Le type de l'attribut est indiqué par le nom d'un des champs du type union. Ce nom sert uniquement à cela.

Ci-contre, l'attribut est de type `S` pour `+` et `X1`, de type entier pour `X2` et `(`.

`NT1` et `NT2` ont un attribut de type `S`, `NT3` et `NT4` ont un attribut entier.

## Définition des types des attributs

- définition de l'union des attributs

```
%union {
  int v;
  struct S {char* c; int n;} s;
}
```

- attributs des lexèmes

```
%token <s> '+' X1
%left <v> X2 '('
```

- attributs des non-terminaux

```
%type <s> NT1 NT2, ...
%type <v> NT3 NT4, ...
```

**Slide 14**

## 3.2 Règles

Ci-contre, `NT` est un non terminal. Les $S_{ij}$ sont des non terminaux ou des lexèmes. L'attribut de `NT` est nommé `$$`, l'attribut du iéme symbole de la partie droite est nommé `$i`. Les attributs sont uniquement synthétisés, c'est à dire qu'on peut calculer la valeur de l'attribut de la partie gauche en fonction d'attributs de la partie droite, mais on ne peut pas calculer la valeur d'un attribut de la partie droite en fonction d'autres attributs de la partie droite, ni en fonction de l'attribur de la partie gauche.

Voici un exemple qui calcule la valeur et le type d'expressions constantes comme `1 + 3 < 2 + 1`.

```
%union {
  int ival;
  struct SVAL {int val; enum {ent, bool} type;} xval;
}

%token <ival> ENTIER
%type <xval> exp
%nonassoc '<'
%left '+'
%%
exp : exp '<' exp    {$$.type = bool; $$.val = ($1.val <  $3.val);}
    | exp '+' exp    {$$.type = ent; $$.val = $1.val + $3.val;}
    | ENTIER         {$$.type = ent; $$.val = $1;}
    ;
```

Les non terminaux s'écrivent comme des identificateurs C. Il est aussi possible de mettre des points au milieu, par exemple `déclaration.de.variable`, **mais pas les tokens** qui sont simplement des identificateurs C, usuellement en majuscules.

### Règles

- Elles sont de la forme générale :

```
NT   : S11 S22 ... { action en C }
     | ...
     ;
```

- Actions : instructions C + notation d'attributs `$i`

```
exp : exp '+' exp    { $$ = $1 + $3 }
    | ...
    | exp '/' exp    { if ($3 != 0)
                         $$ = $1 / $3;
                       else {
                         printf ("division par 0\n");
                         $$ = 0;
```

- L'action est optionnelle, et par défaut c'est `$$ = $1`.

**Slide 15**

Les actions peuvent aussi être insérées au milieu de la partie droite :

```
NT : A { i = 1; } B { $$ = i + $3; } ;
```

mais il ne faut pas se leurrer, cela revient à introduire un nouveau non terminal `A'` et une nouvelle règle $A' \rightarrow \epsilon$. En effet YACC effectue la transformation :

```
NT   : A A' B   { $$ = i + $3; }
A'   :          { i = 1; }
```

avec pour conséquence que cette nouvelle règle peut faire que la grammaire ne soit plus LALR (1). Remarquez aussi que, bien que `B` soit le 2ème symbole de la partie droite, son attribut est `$3`.

Il est possible de donner un n$^o \leq 0$, par exemple `$-3`. En ce cas, on référence un attribut qui est dans la pile en dessous de la partie droite à réduire (`$0` = l'attribut juste en dessous. Il faut être donc sûr de ce qu'il y a dans la pile, et c'est une facilité à éviter en général.

## 3.3 Conflits et précédences

YACC accepte les grammaires ambiguës et les grammaires non LALR(1).

L'utilisation de grammaire ambiguës, notamment pour les expressions, permettent souvent d'écrire la grammaire de manière plus lisible, et réduisent la taille des tables d'analyse (voir exemple ci-dessous).

Les précédences modifient légèrement les règles de résolution des conflits. La précédence d'une règle est celle du dernier lexème de cette règle; certaines règles n'ont donc pas de précédence.

En cas de conflit décalage/réduction, si la règle et la fenêtre ont une précédence, le choix est effectué selon ces précédence (décalage si la fenêtre est plus prioritaire). En cas d'égalité des précédences, l'associativité est prise en compte (associativité gauche implique réduction, associativité droite décalage et non associativité implique erreur).

Il n'est pas très sain de garder une grammaire avec des conflits. En tout cas, toujours regarder `y.output` pour vérifier comment YACC a résolu le conflit.

### Résolution des conflits LALR(1)

- S'il y a des conflits LALR(1), règles appliquées par défaut :
  - conflit décalage/réduction (shift/reduce) : effectuer le shift.
  - conflit réduction/réduction (reduce/reduce) : effectuer la réduction par la règle définie la première parmi les règles en conflit.

```
exp : exp '-' exp
    | exp '*' exp
    | NOMBRE
```

`3 - 2 * 4` → analysé comme `(3 - 2) * 4`

- Les conflits peuvent être levés grâce aux précédences des lexèmes

```
%left '-'
%left '*'
```

**Slide 16**

## 3.4 Automate LALR

Tout ne se règle pas par les priorités des opérateurs. Il faut parfois changer la grammaire.

### Automate LALR

- il est produit sur `y.output` par l'option `-v`
- exemple avec la grammaire

```
expr : var '=' expr
     | var
     | '(' expr ')'
     ;
var : ID
    | var '[' expr ']'
    | '*' expr
    ;
```

- appel à yacc :

```
yacc -v nonlalr.y
conflicts: 2 shift/reduce
```

**Slide 17**

## Automate LALR

– automate : conflit dans l'état 2 (le • des items LR est noté _)

```
2: shift/reduce conflict (shift 6, red'n 2) on =
2: shift/reduce conflict (shift 7, red'n 2) on [
state 2
        expr :  var_= expr
        expr :  var_    (2)
        var :  var_[ expr ]

        =  shift 6
        [  shift 7
        .  reduce 2
```

– `var` en pile, `=` ou `[` en fentre : soit réduire `var` à `expr`, soit décaler

– le décalage (action par défaut) semble correct ici

**Slide 18**

## Automate LALR – 2

– mais l'état 5, qui marque la rencontre de `*`, va sur l'état 2

```
state 5
        var :  *_expr

        ID  shift 4
        (  shift 3
        *  shift 5
        .  error

        expr  goto 9
        var  goto 2
```

– donc `* t[x]` sera analysé comme `* (t[x])` (correct),

– mais `*p = x` sera analysé comme `* (p = x)` (incorrect)

**Slide 19**

Pour bien comprendre le conflit, voici les autres parties intéressantes de l'automate

```
state 0
        $accept : _expr $end

        ID  shift 4
        (  shift 3
        *  shift 5
        .  error

        expr  goto 1
        var  goto 2

state 1
        $accept :  expr_$end

        $end  accept
        .  error

state 3
        var :  ID_    (3)

        .  reduce 3

state 4
        var :  *_expr

        ID  shift 3
        *  shift 4
        .  error

        expr  goto 6
        var  goto 2
```

En fait, la grammaire est ambigu.

## Automate LALR – 3

- il faut soit changer la grammaire

```
expr : var '=' expr | var
     ;
var  : v | '*' var
     ; /*  reduction avec = en fentre */

v    : ID | '(' expr ')' | v '[' expr ']'
     ;  /* decalage avec [ en fentre */
```

- soit garder la grammaire originale avec :

```
%right '='
%right '*'
%left '['
%%
expr : var '=' expr
     | var               %prec '*'
     | '(' expr ')'
     ;
var  : ID | var '[' expr ']' | '*' expr
     ;
```

**Slide 20**

## 3.5 Reprise sur erreurs

L'analyseur syntaxique (fonction `int yyparse ()` retourne 0 si la phrase a été reconnue, 1 si une erreur a été rencontrée.

Deux macros, mises dans une action, permettent de forcer le retour de `yyparse`

- `YYACCEPT` retour avec la valeur 0 (succès)
- `YYABORT` retour avec la valeur 1 (échec)

La macro `YYRECOVERING` retourne 1 lorsque l'analyseur est en phase de reprise sur erreur.

`yyclearin` permet de sauter la fenêtre après une reprise sur erreur. La fenêtre courante est rangée dans la variable `yychar`. S'il n'y a pas de fenêtre courante (immédiatement après un décalage par exemple, où la fenêtre est mise en pile, mais le prochain token pas encore lu), la valeur de `yychar` est `YYEMPTY`.

### Reprise sur erreurs

- quand une erreur est détectée, la fonction `yyerror (char*)` est appelée
- par défaut, le paramètre est `"parse error"`
- Bison permet d'avoir des messages plus précis en définissant dans la partie C des déclarations

```
#define YYERROR_VERBOSE 1
```

- si aucune forme de reprise d'erreur n'est donnée, l'analyse s'arrête
- reprise d'erreur = mode *panique* : on avance dans la phrase jusqu'à un certain caractère, et on remet la pile en état

```
ligne : expression '\n'        { ... }
      | expression error '\n'  { yyerrok; }
```

**Slide 21**

# 4 Couplage Lex Yacc

Yacc est un générateur d'analyseur syntaxique, qui travaille sur des grammaires plus puissantes que les grammaires régulières. Toutefois, pour reconnaître les symboles de base d'un langage (mots clés, opérateurs, constantes, ...), les grammaires régulières sont généralement suffisantes. Aussi les analyseurs syntaxiques ont-ils tendance à se décharger de ce travail sur un analyseur lexical.

En fonctionnement normal, Lex boucle sur le texte jusqu'à la fin de ce texte. Mais un analyseur lexical rend généralement la main à l'analyseur syntaxique après avoir reconnu un symbole élémentaire (appelé aussi "token", symbole terminal, ou encore lexème). Les actions doivent donc se terminer par un code du genre :

```
return code_du_token
```

sauf pour ce qui ne concerne pas l'analyseur syntaxique (les commentaires, les espaces par exemple).

Les terminaux littéraux (par exemple "+") sont codés en YACC par leur code ASCII. Il suffit de retourner ce code (par exemple : return '+';). Les terminaux symboliques (définis dans YACC par `%token`, `%left`, ...), sont codés par YACC avec des entiers $> 255$. YACC peut produire à la demande un fichier `y.tab.h` (En Bison, c'est `nom.tab.h` si le source est `nom.y` constitué de suites de :

```
#define nom_du_terminal code_interne.
```

avec une définition par terminal. Il suffit d'inclure ce fichier dans le source Lex (dans la partie des définitions réservée au code C) pour manipuler les terminaux avec les mêmes noms que YACC (par exemple : `return IDENT;` ).

Dans Yacc, les terminaux peuvent avoir des attributs calculés par Lex. En ce cas, la valeur de l'attribut doit être rangée dans la variable de nom `yylval`, de type `YYSTYPE`.

## Couplage Lex-YACC

- les déclarations de token donnent une macro dans `y.tab.h` (`nom.tab.h` pour Bison), par exemple `%token ID` donne `#define ID 256`.
- inclure `y.tab.h` dans le source Lex
- Les actions doivent retourner le code du token

```
[A-Za-z]+   return ID;
```

- les tokens peuvent calculer des attributs, par exemple en YACC

```
%union {
   int ival;
   ...

%token<ival> NOMBRE
```

en Lex, l'attribut d'un token est calculé dans la variable globale `yylval`

```
[0-9]+  { yylval.ival = atoi (yytext);
          return NOMBRE; }
```

**Slide 22**

# 5 Exemple : une calculette à registres

Cet exemple simule une calculette disposant de 26 registres nommés par une lettre.

## 5.1 Source Yacc

Les déclarations

```
%union {
   int  val;
   char let;
}

%token <val> NUM  /* attribut d'un nombre = valeur entiere *
%token <let> ID   /* l'attribut d'un registre = caractere */
%token FIN        /* marque la fin d'une expression */

%left '+' '-'
%left '*' '/'     /* * et / plus prioritaires que + et - */
%nonassoc UNA     /* pseudo token pour une priorite locale *

%type <val> exp   /* attribut d'une expr = valeur entiere */

%{                /* ici, c'est du code C tout simplement */
static int tab [26];  /* les 26 registres */
%}
```

**Slide 23**

## Les règles de grammaire – 1

```
%%

cal : lig             /* axiome */
    | cal lig
    ;

lig : FIN             { printf ("? ");  }
    | exp FIN         { printf ("= %d\n? ", $1);  }
    | ID '=' exp FIN  { printf ("%c = %d\n? ", $1, $3);
                        tab [$1 - 'a'] = $3;
                      }
    | ID error FIN    { yyerrok; }
    ;
```

**Slide 24**

## Les règles de grammaire – 2

```
exp : NUM                { $$ = $1; }
    | ID                 { $$ = tab [$1 - 'a']; }
    | '-' val %prec UNA  { $$ = - $2; }
    | exp '+' exp        { $$ = $1 + $3; }
    | exp '-' exp        { $$ = $1 - $3; }
    | exp '*' exp        { $$ = $1 * $3; }
    | exp '/' exp        { if ($3 == 0) {
                             yyerror ("division par 0");
                             $3 = 9999;
                           } else
                             $$ = $1 / $3;
                         }
    | '(' exp ')'        { $$ = $2; }
    ;
```

**Slide 25**

## 5.2 Source Lex

### Les fonctions C de YACC

```
%%
int main () {
  printf ("? "); return yyparse ();
}
void yyerror (char*) {
  printf ("erreur de syntaxe\n");
}
```

### Source Lex

```
%{
#include "y.tab.h"
%}
%%
[0-9]+ { yylval.val = atoi (yytext); return NUM; }
[a-z]  { yylval.let = yytext [0]; return ID;]
\n     return FIN;
[ \t]  /* on avale sans rendre la main */
.      return yytext [0];
```

**Slide 26**

# 6 Exercices

1. Donnez les spécifications Lex pour remplacer tous les espaces (blanc, tabulation) d'un texte par un seul espace. Les espaces en fin de ligne seront supprimés.
2. Donnez l'expression régulière des nombres réels, comme `3.14`, `-7.23`, `.15e10`, `50.E-2`. Vérifiez avec Lex (le nombre sera évalué à l'aide de la fonction C `atof`).
3. Donnez l'expression régulière d'un commentaire C : `/* ... */`;
   - attention à `/********/`;
   - attention à `/* commentaire 1*/ du texte C /* commentaire 2 */ ;`

   vérifiez avec Lex.
4. Reconnaître des commentaires C, mais en utilisant `input`
5. Reconnaître des commentaires C, mais avec des start condition.
6. Écrire une grammaire YACC pour reconnaître le langage $a^n b^n$
7. Adapter la calculette pour qu'elle prenne des nombres réels
8. Ajouter à la calculette une expression conditionnelle comme en C

   ```
   b < 3 ? 4 : x + 1
   ```
9. Ajouter à la calculette des fonctions mathématiques usuelles

   ```
   x = sin (t)
   ```