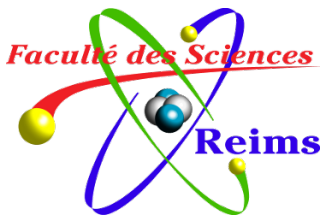
Lex & Yacc

Cyril Rabat
cyril.rabat@univ-reims.fr

Licence 3 Informatique - Info0602 - Langages et compilation

2019-2020



Cours n°5

Présentation des outils Lex & Yacc

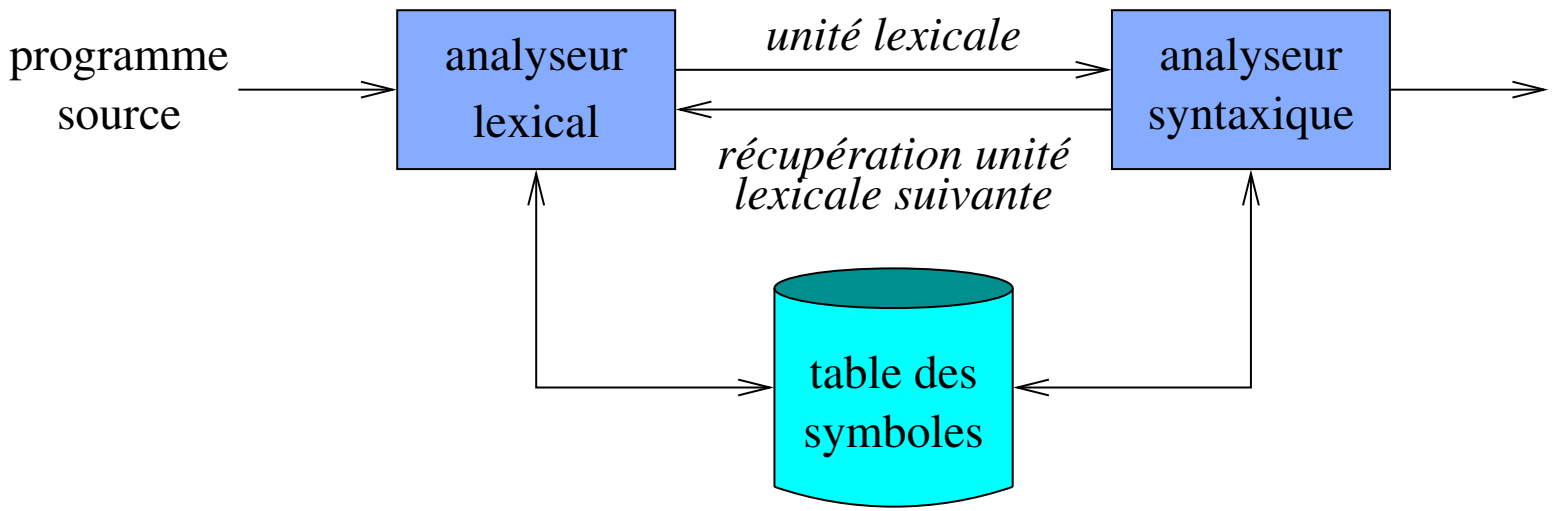
Version 3 mars 2020

Table des matières

5 *Lex&Yacc*
- Introduction
- *Lex*
- *Yacc*
- *Lex&Yacc*

Introduction

- Pour rappel, l'analyseur lexical peut être vu comme un outil pour l'analyseur syntaxique



Le couple Lex&Yacc

- *Lex* (ou *Flex*) est un générateur d'analyseurs lexicaux
  $\hookrightarrow$ Dans la suite, nous parlerons de *Lex*
- *Yacc* (ou *Bison*) est un générateur d'analyseurs grammaticaux
  $\hookrightarrow$ Dans la suite, nous parlerons de *Yacc*
- Ces outils sont généralement utilisés en couple

Attention

Il existe des différences entre *Lex* et *Flex*, et *Yacc* et *Bison*

Unité lexicale, modèle, lexème et attribut (rappels)

- **Modèle** : règle qui décrit un ensemble de chaînes
- **Unité lexicale** : produite par l'ensemble de chaînes du modèle
  $\hookrightarrow$ Exemple : mots-clés, opérateurs, identificateurs, constantes, chaînes littérales...
- **Lexème** : la suite de caractères du programme source qui correspond au modèle

Exemple

- Unité lexicale : *chiffre*
- Lexèmes : 0, 1, 2
- Modèle : $[0-9]$

- **Attributs** : données liées aux unités lexicales
  $\hookrightarrow$ Exemple : l'entrée dans la table des symboles pour un identificateur

Présentation de *Lex*

- *Lex* est un générateur d'analyseurs lexicaux
  $\hookrightarrow$ *Flex* est la version GNU
- Il permet de définir un ensemble d'unités lexicales
- Chacune est décrite par une expression régulière (le modèle)
- Il produit un automate fini
- Il est possible d'associer du code C aux unités lexicales

Remarques

- Préférez l'usage de *Flex* pour une compatibilité avec ce cours

Structure générale du fichier XX.lex

```
%{
/* Les déclarations en C */
%}

/* Définitions */

%%

/* Règles */

%%

/* Fonctions C */
```

Caractères spéciaux

- Tous les caractères sont significatifs
- Il existe des caractères spéciaux : utilisation de \ pour les échapper

Caractère	Signification	Exemple	Produit
+	1 ou plus	a+	a...a
*	0 ou plus	a*	ϵ ou a...a
?	0 ou 1 fois	a?	ϵ ou a
|	union	a|b	a ou b
(...)	factorisation	(a|b)c	ac ou bc

- Priorités :
  - +, *, ?
  - concaténation (abc)
  - union (|)

Autres caractères spéciaux (1/2)

Car.	Signification	Exemple	Produit
"	valeur littérale des caractères	`"+?"+`	`+?...+?`
\	valeur littérale des caractères	`\+\?+`	`+?...?`
.	tout sauf fin de ligne	`.|\n`	tous les caractères
`[...]`	ensemble de caractères	`[01]`	0 ou 1
-	intervalle	`[a-z]`	les lettres
^	complément	`[^0-9]`	tout sauf chiffre

Remarques

- Attention aux expressions régulières entre crochets
  ↪ Exemple : `[(0|1)+]` signifie `(`, `0`, `|`, `1`, `)` ou `+`

Autres caractères spéciaux (2/2)

Car.	Signification	Exemple	Produit
`{...}`	Nom d'une expression régulière	`{CHIFFRE}`	chiffres
	Occurrence	`a{1,5}`	entre 1 et 5 `a`
		`a{2,}`	2 `a` ou plus
		`a{3}`	3 `a`
`$`	reconnaissance en fin de ligne	`" "$`	termine par " "
`^`	reconnaissance en début de ligne	`^" "`	commence par " "

Classes de caractères

- Un ensemble de classes de caractères est définie (utilisables en C)
  ↪ Permet de simplifier les expressions régulières
- Syntaxe : `[:X:]` où X est la classe
- Liste des classes :
  - `[:alpha:]` : caractères alphabétiques
  - `[:digit:]` : digit (`[0-9]`)
  - `[:alnum:]` : caractères alphanumériques (équivalent à `[:alpha:]|[:digit:]`)
  - `[:cntrl:]` : caractères de contrôle
  - `[:graph:]` : caractères imprimables (sauf espace)
  - `[:lower:]`, `[:upper]` : caractères minuscules ou majuscules
  - `[:print:]` : caractères imprimables (avec l'espace)
  - `[:punct:]` : caractères imprimables sauf alphanumériques et espace
  - `[:space:]` : espace ; peut contenir `\t`, `\n`, `\r`, `\v`
  - `[:xdigit:]` : caractères hexadécimaux

Définitions d'identificateurs

- Simplifient l'écriture des expressions régulières
- Placées avant le premier %%
- Séparation entre l'identificateur et l'expression régulière : espace, tabulation

Exemple : un réel

```
chiffre   [0-9]
entier    {chiffre}+
reel      {entier}(\.{entier})?
```

Règles

- Composées d'expressions régulières et éventuellement d'actions (en C)
- Les actions sont des instructions ou des blocs d'instructions (entre accolades)
- Si rien, pas d'action
- L'action `|` signifie qu'il faut faire l'action de la ligne suivante
- Le lexème est stocké dans la variable `yytext`
  ↪ *Flex* : `char yytext[]`
  ↪ *Lex* : `char* yytext`
- La longueur du lexème est stockée dans `yyleng` (un `int`)
- Par défaut, une chaîne non reconnue est affichée sur la sortie
- Une fois une chaîne reconnue :
  - L'action correspondante est exécutée
  - La suite du texte est analysée

Règles : exemple

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

int numLigne = 1;

%}

CHIFFRE [0-9]

%%

{CHIFFRE} printf("Chiffre %s (sur la ligne %d)\n", yytext, numLigne);
\n        numLigne++;
.         ;

%%
```

Choix des règles

- L'ordre de définition des règles est important
- Par défaut, la règle qui correspond à plus de caractères est sélectionnée
- Si la même chaîne est reconnue par deux règles, c'est la première définie qui est choisie
  ↪ La macro `REJECT` (dans les actions) permet de passer à la règle suivante
- Règle par défaut :
  ↪ `. | \n ECHO` : tout ce qui n'est pas reconnu est affiché
  ↪ Ajoutez la règle `. ;` pour ignorer les caractères inconnus

Exemple

```
CHIFFRE [0-9]
ENTIER {CHIFFRE}+
REEL {ENTIER}(\.{ENTIER}+)?

%%

{ENTIER} printf("Entier %d\n", atoi(yytext));
{REEL}   printf("Réel %f\n", atof(yytext));
\n       numLigne++;
.        ECHO;
```

Sortie

- Saisie “1234” donne `entier`
- Saisie “1234.1234” donne `réel`

Fonctions et macros (1/2)

- La dernière partie peut contenir des fonctions C
  ↪ Recopiées telles quelles dans `lex.yy.c` (généré lors de la compilation avec *Flex*)
- Un ensemble de fonctions sont définies par défaut
- L'entrée est réalisée sur `yyin` (par défaut `stdin`)
- La sortie est réalisée sur `yyout` (par défaut `stdout`)
- Possible de les redéfinir
  ↪ `yyin` sur un fichier, par exemple

`yywrap`

- Appelée une fois l'analyse terminée
- Par défaut, retourne 1, mais peut être redéfinie :
  ↪ Pour changer de fichier d'entrée, par exemple (`yyin`)

Fonctions et macros (2/2)

`ECHO`

- Affiche le contenu de `yytext` à l'écran
  ↪ `printf("%s", yytext)`

`input(c)`

- Lit le prochain caractère dans le flux d'entrée (`yyin`)

`unput(c)`

- Replace le caractère dans le flux d'entrée
- Pour replacer tout `yytext`, le faire en sens inverse !

`yyless(n)`

- Retour de `yyleng-n` caractères en arrière
- Les autres caractères sont replacés dans le flux d'entrée

Programme complet

```
%{
#include <stdio.h>
int numLigne = 1;
%}

CHIFFRE [0-9]
ENTIER {CHIFFRE}+
REEL {ENTIER}(\.{ENTIER}+)?

%%

{ENTIER} printf("Entier %d\n", atoi(yytext));
{REEL}   printf("Réel %f\n", atof(yytext));
\n       numLigne++;
.        ;

%%

int main() {
  yylex();
  return EXIT_SUCCESS;
}
```

Pour conclure

- Tout n'a pas été présenté ici
  $\hookrightarrow$ Par exemple : les contextes
- Nous n'utiliserons qu'une partie des fonctionnalités de *Lex*
- Un TP est prévu pour l'usage de *Lex* seul ...
- ... mais nous l'utiliserons préférentiellement avec *Yacc*

Présentation de Yacc

- *Yacc* est un générateur d'analyseurs grammaticaux
  ↪ *Bison* est la version GNU
- Il peut interagir avec *Lex* même si ce n'est pas obligatoire
- Différents paramètres :
  - `-v` : produit les tables d'analyse dans le fichier `y.output`
  - `-d` : produit un fichier `y.tab.h` pour être utilisé dans l'analyseur lexical
    ↪ Définitions des lexèmes et des types des attributs

Structure générale du fichier XX.yacc.y

```
%{
/* Includes */
%}

/* Définitions */

%%

/* Règles */

%%

/* Fonctions C */
```

Définitions

- Partie contenant les déclarations des lexèmes, de l'axiome, les types des symboles (terminaux ou non)
- Les terminaux :
  - Déclarés à l'aide de `%token`
    ↪ Exemple : `%token ENTIER REEL ID WHILE IF THEN`
  - Inutile de déclarer les lexèmes simples (comme les opérateurs)
- Déclaration de la priorité et de l'associativité :
  - Permet d'éviter les conflits dans la grammaire
  - Utilisation de `%left`, `%right` et `%nonassoc`
  - La priorité dépend de l'ordre de déclaration (moins prioritaires en premier)
  - Inutile de déclarer les lexèmes avec `%token`

Remarque

Il est possible de modifier temporairement la priorité directement dans une règle avec `%prec`

Exemple

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

%token ENTIER
%left '+', '-'
%left '*', '/'

%%

expression: ENTIER
      | expression '+' expression
      | expression '*' expression
      ;

%%

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

Définitions : déclaration de l'axiome

- Non obligatoire :
  $\hookrightarrow$ La première règle définit l'axiome (partie gauche)
- Syntaxe : `%start expression`

Définitions : types des symboles (1/2)

- Par défaut, les symboles sont de type entier
- Type défini par la constante `YYSTYPE`
  ↪ Recopiée dans `y.tab.h` avec l'option `-d`
- Possible de redéfinir les types possibles dans une union

Exemple

```
%union {
  int entier;
  float reel;
  struct {
    char nom[16];
    char prenom[16];
  } personne;
};
```

Définitions : types des symboles (2/2)

- Pour définir un type, utilisation de < et > après `%token`, `%left`, `%right`
- Le type correspond à un champ de l'union

Exemple

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

%token<entier> ENTIER
%token<reel> REEL
```

Règles

- Règles de la forme suivante :

```
exp : A B { /* Actions */}
    | C { /* Actions */}
    ;
```

- Ici, `exp` est un non terminal, `A`, `B` et `C` sont des symboles terminaux ou non
- La dernière ligne correspond à $exp \rightarrow \epsilon$
- Dans le code en C (actions), `exp` correspond à `$$`, `A` à `$1` et `B` à `$2`
- La partie code est optionnelle ; par défaut elle vaut `$$ = $1;`
- Le code est exécuté lorsque la règle est réduite

Exemple

```
%token ENTIER

%%

programme: expression '.' {
    printf("=%d\n", $1);
  }
         |
         ;

expression: ENTIER
      | expression '+' expression {
        $$ = $1 + $3;
      }
      | expression '-' expression {
        $$ = $1 - $3;
      };
```

Actions au milieu des règles

- Il est possible d'introduire du code au milieu de la partie droite de la règle
- Par exemple :

```
exp : exp1 { /* Actions */} '+' exp2 { /* Actions */}
```

- La valeur de `exp2` est `$4` (et non `$3`)
- A noter que *Yacc* effectue la transformation suivante :

```
exp : exp1 exp3 '+' exp2 { /* Actions */}
exp3 : { /* Actions */}
```

- Cela peut faire que la grammaire ne soit plus LALR(1)

Grammaires acceptées par *Yacc*

- *Yacc* accepte des grammaires ambiguës et non LALR(1)
  $\hookrightarrow$ Affichage des conflits
- L'utilisateur peut régler les conflits à l'aide des précédences
- Si des conflits décalages/réduction, *Yacc* exploite les précédences
- Il est possible de vérifier dans `y.output` comment il les gère

Attention pour Info0602

Vous ne devez jamais produire de grammaires qui possèdent des conflits.

Exemple de conflit décalage/réduction

```
%%

expression: ENTIER
      | expression '+' expression
      | expression '*' expression
      ;
```

`1 + 2 * 3` analysé comme $(1 + 2) * 3$

Exemple de conflit décalage/réduction : `y.output`

```
État 6 conflits: 2 décalage/réduction
État 7 conflits: 2 décalage/réduction
...
état 6

    2 expression: expression . '+' expression
    2           | expression '+' expression .
    3           | expression . '*' expression

    '+'  décalage et aller à l'état 4
    '*'  décalage et aller à l'état 5

    '+'        [réduction par utilisation de la règle 2 (expression)]
    '*'        [réduction par utilisation de la règle 2 (expression)]
    $défaut  réduction par utilisation de la règle 2 (expression)
```

Exemple de conflit décalage/réduction : résolution

```
%left '+'
%left '*'

%%

expression: ENTIER
      | expression '+' expression
      | xexpression '*' expression
      ;
```

Gestion des erreurs

- Dès qu'une erreur est détectée, fonction `yyerror` appelée
- Possible de la redéfinir
- Par défaut, `int yyparse()` :
  ↪ Retourne 0 si la phrase a été reconnue ou 1 en cas d'erreur
- Deux macros modifient le retour (dans des actions) :
  - `YYACCEPT` : acceptation
  - `YYABORT` : échec
- La macro `YYRECOVERING` vaut 1 si l'analyseur est en phase de reprise sur erreur
- Pour sauter la fenêtre après reprise sur erreur : `yyclearin`
  ↪ `yychar` contient alors la fenêtre
  ↪ Il faut `YYEMPTY` s'il n'y a pas de fenêtre courante

Couplage de *Lex* et de *Yacc*

- *Lex* est appelé par *Yacc*
  ↪ Il retourne les terminaux
- Les actions des règles se terminent par des `return`
  ↪ *Yacc* peut récupérer les symboles analysés
- Il est possible de retourner des valeurs :
  ↪ Exemple : le symbole `ENTIER` correspond à une valeur (lexème)
- Utilisation d'une variable `yylval` de type `YYSTYPE`