Systèmes d'exploitation

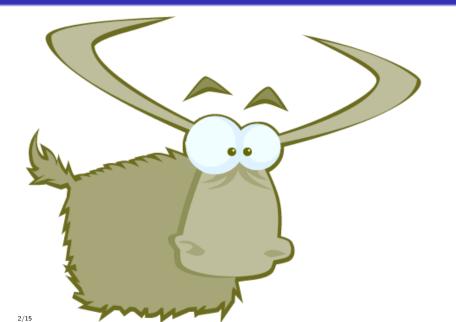
Lex et Yacc : partie II (Yacc)

Yves STADLER

Codasystem, UPV-M

13 septembre 2011

Introduction



Introduction

Yet Another Compiler Compiler

Analyseur grammatical

Historique

- Stephen C. Johnson at AT&T
- Maintenant nommé bison

Introduction

Principe

- Décrire des tokens (éléments de langages)
- Écire une liste de règles qui décrivent un langage (Comment s'organisent les tokens)
- Générer un programme qui va comprendre un langage
- Une calculatrice comprends le langage des expressions mathématiques
- Le fichier .yy donnera une source dans le langage choisi

Général

- Pareil qu'un fichier Lex
- Definition section %%Rule section

%%
Additionnal code

• Le fichier aura généralement l'extension .yy

Definitions

- Includes comme en lex (%{ ... %})
- Définition de tokens
- Un token est un élément de syntaxe par exemple PLUS (%token PLUS)
- Un token ne réprésente rien! Il n'a aucune valeur! Il a juste un sens!
- Exemple le token PLUS ne correspond pas à la chaîne "+"
- Le token PLUS correspond au sens de l'addition (On expliquera son contexte d'utilisation dans la section suivante)
- Un axiome (%axiom start) est le nom de l'axiome de la grammaire. Par défaut c'est la première règle
- Supposons que nous avons les tokens suivants : NOMBRE PLUS MOINS FOIS DIVISE PUISSANCE

Productions

- Donner un sens à nos token
- notion_grammaticale:
 notionOuToken1 {action sémantique}
 | notionOuToken2 {action sémantique}
 ...
 | notionOuToken3 {action sémantique}
 ...
- NotionOuTokeni est soit une autre production/notion de cette section, soit un token de la section précédente. (Peut être composé)
- Les actions sont effectuées lorsque YACC réussi à donner un sens à le notionOuToken qui précède.

Exemple : expressions mathématiques simples

```
EXPRESSION: NOMBRE
| EXPRESSION PLUS EXPRESSION {//calcul de l'addition}
| EXPRESSION MOINS EXPRESION {//calcul de l'addition}
| EXPRESSION FOIS EXPRESION {//calcul de l'addition}
| EXPRESSION DIVISE EXPRESION {//calcul de l'addition};
```

Lecture

- On dit qu'une expression est décrite par :
- Une addition de deux autres expressions
- Une soustraction de deux expressions
- Une multiplication de deux expressions
- Une division de deux expressions

Associativité

Dans notre cas EXPRESION PLUS EXPRESSION il y a ambiguité, YACC ne sait pas s'il doit chercher d'abord la première expression ou d'abord la seconde.

- On résoud le problème en précisant l'associativité de PLUS
- %left PLUS va dire que l'on commence par la gauche (par exemple pour une addition)
- %right PUISSANCE va dire que l'on commence par évaluer a right (par exemple pour une puissance)
- On peut rendre un token non associatif : %nonassoc EGAL

Précédence

YACC ne sait pas non plus dans une expression composée (ex : 2 + 3 * 5) s'il doit commencer d'abord par la règle PLUS ou la règle FOIS

- On indique grâce à %left et %right la priorité des opérateurs
- Première ligne moins prioritaire
- Dernière ligne plus prioritaire

%left PLUS MOINS

%left FOIS DIVISE

Code additionnel

On peut comme en lex y définir certaines fonctions dont

```
• int yyerror(char *s) {
    printf("%s\n",s);
}

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

Principe

- Utiliser la reconnaissance grammaticale pour donner un sens à l'analyse lexicale.
- Exemple créer un programme C qui va pouvoir interpréter nos expressions mathématique et calculer le résultat
- Variable commune: yylval de type YYSTYPE (on pourra faire un define YYSTYPE double par exemple.)

```
Exmple: fichier lex
         [\t]+
SPACE
     [0-9]
DIGIT
      {DIGIT}+
INT
     {DIGIT}("."{DIGIT})?
FLOAT
%%
{SPACE} { /* Ignorés */ }
{FLOAT} { yylval=atof(yytext);
        return(NOMBRE);
"+"
      return(PLUS);
11 _ 11
      return(MOINS);
"*"
      return(FOIS);
"/"
      return(DIVISE);
"\n"
      return(FIN);
```

Exemple: fichier yacc

- On a déjà vu comment définir les tokens et leurs priorités
- On choisit %axiom LIGNE
- On met aussi les règles que l'on a choisis
- Comment faire maintenant pour calculer quelque chose?

Variable

- notion: TOKEN1 TOKEN2 TOKEN3
- \$\$ représente la valeur de la notion
- \$i représente la valeur du token i
- EXPRESSION: NOMBRE {\$\$=\$1} expression va prendre la valeur du token (calculer par la reconnaissance lex)
- EXPRESSION: EXPRESSION PLUS EXPRESSION {\$\$=\$1 + \$2} expression va prendre la valeur de la 1er plus la seconde expression (Calculés par une autre notion YACC)

Exemple

```
EXPRESSION: NOMBRE {$$=$1}
| EXPRESSION PLUS EXPRESSION {$$=$1+$3}
| EXPRESSION MOINS EXPRESSION {$$=$1-$3}
| EXPRESSION DIVISE EXPRESSION {$$=$1/$3}
| EXPRESSION FOIS EXPRESSION {$$=$1*$3}
;
LIGNE: LIGNE FIN {printf("Le resultat est %f\n", $1);}
| FIN
| error FIN {yyerrok; /* Rattrapage d'erreur */}
```