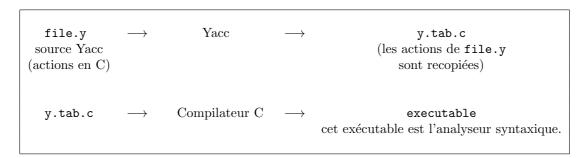


Master d'informatique 1ère année Compilation

Présentation de Yacc

1 Schéma d'utilisation de Yacc

Yacc (Yet Another Compiler Compiler) est un constructeur d'analyseur LALR. Il fonctionne de la façon suivante.



Le fichier de départ file.y contient une spécification du traducteur pour Yacc. La commande yacc file.y produit un programme y.tab.c. Ce programme est une implantation d'analyseur LALR écrite en C.

En compilant y.tab.c, nous obtenons un exécutable qui effectue la traduction spécifiée par le programme Yacc.

Donc un fichier y.tab.c est un programme C comprenant :

- une représentation compressée des tables d'analyse LALR,
- une routine standard d'utilisation de cette table pour simuler le fonctionnement de l'automate,
- les actions de y.tab.c recopiées.

2 Description d'un source Yacc

Un source Yacc contient

- l'ensemble des règles d'une grammaire,
- des actions, associées aux règles, qui seront exécutées lorsque la règle correspondante est reconnue,
- une procédure d'analyse lexicale qui détermine les tokens.

La forme générale d'un source Yacc est la suivante :

déclarations <- optionnel %%
règles de traduction
%% <- optionnel procédures auxilliaires <- optionnel

Les espaces, tabulateurs et sauts de lignes sont ignorés. Les commentaires, similaires à ceux de C /*...*/ peuvent apparaître n'importe où dans le source.

Les symboles terminaux ou tokens sont soit définis dans la partie déclaration, soit placés entre apostrophes ('...'). Toutes les autres chaines de formées de lettres et/ou de chiffres non entourées d'apostrophes et non déclarées comme des tokens représentent des symboles non terminaux. Tout non terminal doit apparaitre à gauche d'au moins une des règles.

2.1 Les déclarations (optionnelles)

On y trouve des déclarations de variables globales (comprises entre $\{\ldots,\}$), ainsi que diverses définitions spécifiques à Yacc comme :

- les déclarations de tokens : %token nom1 nom2 ...
- la déclaration du symbole initial S de la grammaire :
 %start S. Lorsque cette déclaration est absente, le symbole non terminal apparaissant à gauche de la première règle est par défaut considéré comme le symbole initial.
- le type d'associativité d'un symbole : %right op ou %left op ou %nonassoc op pour un opérateur op respectivement associatif à droite, à gauche ou non associatif.

2.2 Règles de traduction

Chaque règle est formée d'une règle de production de la grammaire et de son action sémantique associée. Un ensemble de règles < partie $gauche> \rightarrow < alt_1> | < alt_2> | ... | < alt_n> s'écrit en Yacc :$

Une action sémantique est une séquence d'instructions C. Elle est associée à une règle de la grammaire et est exécutée à chaque fois que la règle correspondante est reconnue et réduite. Dans une action sémantique, le symbole \$\$ référence la valeur de l'attribut associé au non terminal de la partie gauche, tandis que \$i référence la valeur associée avec le i-ème symbole de la grammaire (terminal ou non terminal) en partie droite.

2.3 Procédures auxiliaires (optionnelles)

Ce sont des routines C qui aident à la traduction. Un analyseur lexical de nom yylex() doit être fourni, soit directement dans cette section, soit produit directement par Lex. Il doit retourner les unités syntaxiques ou tokens déclarés dans la première section.

2.4 Exemple d'utilisation

Construisons un calculateur de bureau simplifié qui lit une expression arithmétique, l'évalue et l'imprime. Nous allons construire ce calculateur de bureau en partant de la grammaire suivante :

```
E -> E + T | T
T -> T * F | F
F -> (E) | ENTIER
```

où ENTIER désigne un symbole terminal.

Une solution Yacc est la suivante :

```
%{
#include <stdio.h>
%}
%token ENTIER
%start ligne
%%
ligne : E '\n' {printf ("%d\n",$1);}
E : E '+' T
                \{\$\$ = \$1 + \$3; \}
  | T
T : T '*' F
                \{\$\$ = \$1 * \$3; \}
F : '(' E ')' {$$ = $2; }
  | ENTIER
%%
yylex ()
   int c;
```

```
while ((c = getchar()) == ' ') {/* sauter blancs */}
if (isdigit (c)) {
    yylval = c - '0';
    return (ENTIER);
    }
return (c);
}
```

Le programme généré par Yacc utilise une routine d'analyse lexicale yylex qui peut être décrite à la fin du source yacc, ou qui peut être engendrée par Lex.

Dans le source Yacc ci-dessus l'action sémantique $\{\$\$ = \$1;\}$ est omise sur les règles $E \to T$, $T\to F$ et $F\to ENTIER$ car elle est réalisée par défaut.

3 Utilisation de Yacc avec des grammaires ambiguës

Lorsque Yacc est en présence d'une grammaire ambiguë, l'algorithme LALR produira des conflits, soit de type shift/reduce, soit de type reduce/reduce. Yacc rendra compte du nombre de conflits détectés. On peut étudier ces conflits en appelant Yacc avec l'option -v. Cette option produit un fichier y.output qui contient une description des états, une représentation lisible de la table LALR et des conflits. Chaque fois que Yacc rend compte de la découverte de conflits, il est conseillé de créer et de consulter le fichier y.output afin d'étudier la raison pour laquelle ces conflits ont été produits et de vérifier qu'ils ont été résolus correctement.

Yacc résout tous les conflits en utilisant deux règles :

- lors d'un conflit reduce/reduce Yacc choisit la production apparaissant en premier dans le source Yacc;
- 2. lors d'un conflit shift/reduce Yacc choisit le shift.

Les grammaires d'opérateurs, comme la grammaire naive des expressions arithmétiques :

```
E -> E + E
| E * E
| (E)
| ENTIER
```

présentent des ambiguités, dues à l'absence de prise en compte, au niveau de la grammaire, des règles de priorité et d'associativité des opérateurs. Ces ambiguités entrainent des conflits qui peuvent être résolus en spécifiant, dans la partie déclaration, les règles et priorité et d'associativité des tokens, avec les instructions "left, "right et "nonassoc, comme"

```
%right '='
%left '+' '-'
%left '*' '/'
```

Tous les tokens sur la même ligne ont les mêmes priorité et associativité. Les lignes sont listées dans l'ordre croissant des priorités.

On peut souhaiter, dans une règle donnée, affecter à un opérateur une priorité particulière qui ne s'applique que dans cette règle. Pour cela on utilise la commande "prec. Ainsi dans la règle

```
A : B op1 C %prec op2
```

l'opérateur op1 a, dans cette règle seulement, la même priorité que l'opérateur op2. Ce mécanisme est utile pour traiter le cas du *moins unaire* qui doit avoir une priorité différente du *moins binaire*, supérieure à celle du *multiplier*.

A chaque règle de grammaire, Yacc associe une priorité et associativité : celles du dernier token de la règle (si elles sont définies).

Dans le cas d'un conflit *shift/reduce* ou *reduce/reduce* pour lequel le token courant et/ou la règle de grammaire considérée n'ont pas de priorité et d'associativité, les règles standards (1) et (2) de désambiguité s'appliquent.

Lorsque le token courant et la règle considérée ont une priorité et une associativité, le conflit est résolu en faveur de l'action *shift* ou *reduce* avec la plus forte priorité. Si les priorités sont identiques, alors Yacc utilise l'associativité. Le *reduce* est retenu en cas d'associativité gauche, le *shift* en cas d'associativité droite, la non associativité conduisant à une erreur.

4 Récupération d'erreurs

Yacc permet de faire une récupération sur les erreurs grâce à l'utilisation du token error réservé à la gestion des erreurs. Ce token peut être ajouté dans une règle de grammaire. Il suggère la place où des erreurs sont attendues et où une récupération peut être effectuée. En cas d'erreur, Yacc dépile les symboles jusqu'à ce qu'il trouve un état où le token error est légal. Il agit alors comme si le token courant était ce token error et exécute l'action spécifiée. Si aucune règle d'erreur n'est spécifiée, l'analyseur s'arrête dès qu'une erreur syntaxique est détectée.

Un exemple utile de traitement des erreurs est donné par la règle suivante.

```
stat : error ';'
```

Quand une erreur apparait lors de l'analyse d'une instruction définie par le non-terminal stat, l'analyseur saute l'instruction jusqu'au ';' suivant. Tous les tokens après l'erreur et avant le ';' suivant sont ignorés. Quand le ';' est rencontré, la règle est réduite et les actions de "nettoyage" associées sont exécutées.

5 Attributs de type quelconque

Les variables Yacc \$\$ et \$i permettent de manipuler des attributs associés aux différents symboles terminaux et non terminaux utilisés dans une règle de grammaire. Lorsque le ième symbole est un terminal, \$i désigne la valeur du token, qui est mémorisée pendant l'analyse lexicale dans la variable yylval.

Pour avoir des attributs de types arbitraires, il faut déclarer ces types au moyen d'une union :

```
%union {
     type1 nom1;
     type2 nom2;
     ...
}
```

où les type sont des types C. Il faut alors déclarer le type de chaque token (ici les tokens t1 et t2) et des symboles non-terminaux (ici S), de la façon suivante :

```
%token <nom1> t1 t2
%type <nom2> S
```

6 Utilisation avec Lex

La variable globale yylval, commune à Lex et Yacc peut être utilisée pour transmettre de Lex à Yacc la valeur associée à un token. Cette valeur est généralement calculée à partir de yytext qui correspond à la suite de caractères extraits par Lex du flux d'entrée. Par défaut, yylval est définie comme un entier par Yacc, il doit alors être déclaré comme externe dans Lex.

fichier lex:

yylval peut être redéfini dans la deuxième partie du fichier yacc grâce à %union. Cela permet de transmettre

de Lex vers Yacc des valeurs dont le type varie selon le "type <nombre> operation symbole reconnu. Il faudra bien évidemment typer les terminaux et/ou non-terminaux de la grammaire Yacc en conséquence en utilisant %type.

fichier lex:

```
%%
. . .
[0-9]+
          {yylval.nombre=atoi(yytext);
           return ENTIER;}
[a-zA-Z]+ {strcpy(yylval.variable,yytext);
           return IDENTIFICATEUR;}
[+]
      {return PLUS;}
fichier yacc:
%union {int nombre; char* variable;}
%token ENTIER IDENTIFICATEUR
```

```
operation : operation PLUS ENTIER {$$=$1+$3;}
          | ENTIER
                            {$$=yylval.nombre;}
          | IDENTIFICATEUR
                   {$$=valeur(yyval.variable);}
Les générations et compilations de l'analyseur lexical et
```

de l'analyseur syntaxique doivent être coordonnées :

- Construire le fichier lex et le fichier yacc
- Lancer Yacc avec l'option -d pour créer y.tab.h : yacc -d file.y
- Rajouter l'inclusion de la définition des tokens dans la première partie du fichier lex : #include "y.tab.h"
- Lancer Lex: lex file.1
- Compiler: gcc lex.yy.c y.tab.c -lfl