Université Mohammed V - Agdal
Faculté des sciences
Département d'Informatique

Cours de Compilation

SMI - S5

Prof. M.D. RAHMANI
mrahmani@fsr.ac.ma

L'analyse ascendante (LR, RR)

Plan:

1- Le but d'un analyseur ascendant

2- Exemple de la méthode LR

3- Implantation d'une grammaire LR

4- Le langage YACC (LALR)

L'analyse ascendante (LR, RR)

1- **But:**

L'***analyse ascendante*** consiste à ***réduire*** la chaîne d'entrée en un symbole initial, qui est l'***axiome***.

Il s'agit de la construction de l'arbre syntaxique en partant des éléments de la phrase à analyser (les terminaux) qui sont les feuilles de l'arbre, vers la racine qui correspond à l'axiome de la grammaire.

L'analyse ascendante (LR, RR)

Remarques:

- Ce processus de construction de l'arbre est une réduction, qui se traduit par le remplacement de la partie droite des productions par la partie gauche.

- Les grammaires LR (ascendantes) sont plus générales que les grammaires LL (descendantes).

En fait pour les algorithmes ascendants, la décision d'effectuer une réduction donnée est prise lorsque l'on dispose de l'ensemble de son membre droit pour le remplacer par le non terminal à gauche,

- Par contre, dans le cas d'un algorithme descendant, la décision est prise lorsque l'on dispose de 1 (k) symbole de son membre gauche.

- Dans le 1er cas, on dispose en général de plus d'informations que dans le second.

L'analyse ascendante LR

2- **Exemple de la méthode LR:**

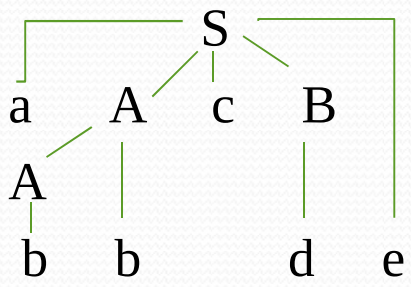
Soit la grammaire:

S ⟶ a A c B e

A ⟶ A b | b

B ⟶ d

Et soit la chaîne à analyser : ω = "abbcde"

chaîne d'entrée	production utilisée	Arbre d'analyse
abbcde	A ⟶ b	
aAbcde	A ⟶ Ab	
aAcde	B ⟶ d	
aAcdBe	S ⟶ aAcBe	

L'analyse ascendante LR

3- **Implantation de la méthode LR:**

Il s'agit d'une méthode d'analyse générale appelée "***analyse par décalage-réduction***" (*Shift Reduce Parsing*)

	Pile	Tampon
état initial:	$	ω$
état final:	$S	$

avec ω la chaîne à analyser et S l'axiome de la grammaire.

A l'état final, la chaîne ω est réduite en l'axiome à moins qu'une erreur est détectée pendant l'analyse.

L'analyse ascendante LR

Les opérations à effectuer sont:

1- `Décalage`: le symbole d'entrée courant est inséré dans la pile,

2- `Réduction`: l'analyseur reconnait la partie de droite d'une production au sommet de la pile, elle la remplace alors par le non-terminal correspondant,

3- `Acceptation`: la chaîne est réduite en l'axiome,

4- `Erreur`: détection d'une erreur de syntaxe et appel d'une routine de traitement d'erreur.

L'analyse ascendante LR

Exemple de fonctionnement:

Pile	Tampon	Action
$	abbcde$	décalage
$a	bbcde$	décalage
$ab	bcde$	réduction A ⟶ b (1)
$aA	bcde$	décalage
$aAb	cde$	réduction A ⟶ Ab (1) (2)
$aA	cde$	décalage
$aAc	de$	décalage
$aAcd	e$	réduction B ⟶ d (1)
$aAcB	e$	décalage
$aAcBe	$	réduction S ⟶ aAcBe
$S	$	ACCEPTATION

L'analyse ascendante LR

Nous avons dans cette méthode d'analyse ascendante 2 types de conflits:

1- Un conflit entre un décalage et un réduction (1),

nous avons choisi dans cet exemple de favoriser la réduction par rapport au décalage.

2- Un conflit entre deux réductions (2),

nous avons choisi, la réduction du membre droit avec le plus grand nombre de symboles.

Les inconvénients:

La méthode n'est pas déterministe et exige un gros travail d'analyse,

Peu de langages de programmation utilisent ces méthodes.

L'analyse ascendante LR

3 méthodes dérivent de la méthode générale:

- ***SLR*** (*Simple* LR): facile à implanter et non ambigüe mais adaptée à une classe limitée de grammaires,

- ***LR canonique***: est la plus efficace car la plus générale mais la plus couteuse en temps et en mémoire,

- ***LALR*** (*Look Ahead*) est intermédiaire entre les 2 méthodes précédentes et prédictive,

La dernière méthode a donné naissance à l'outil ***YACC***

Le langage YACC

Conçu au début des années 70 par Johnson, le langage YACC veut dire, *Yet Another Compiler Compiler (encore un autre compilateur de compilateurs).*

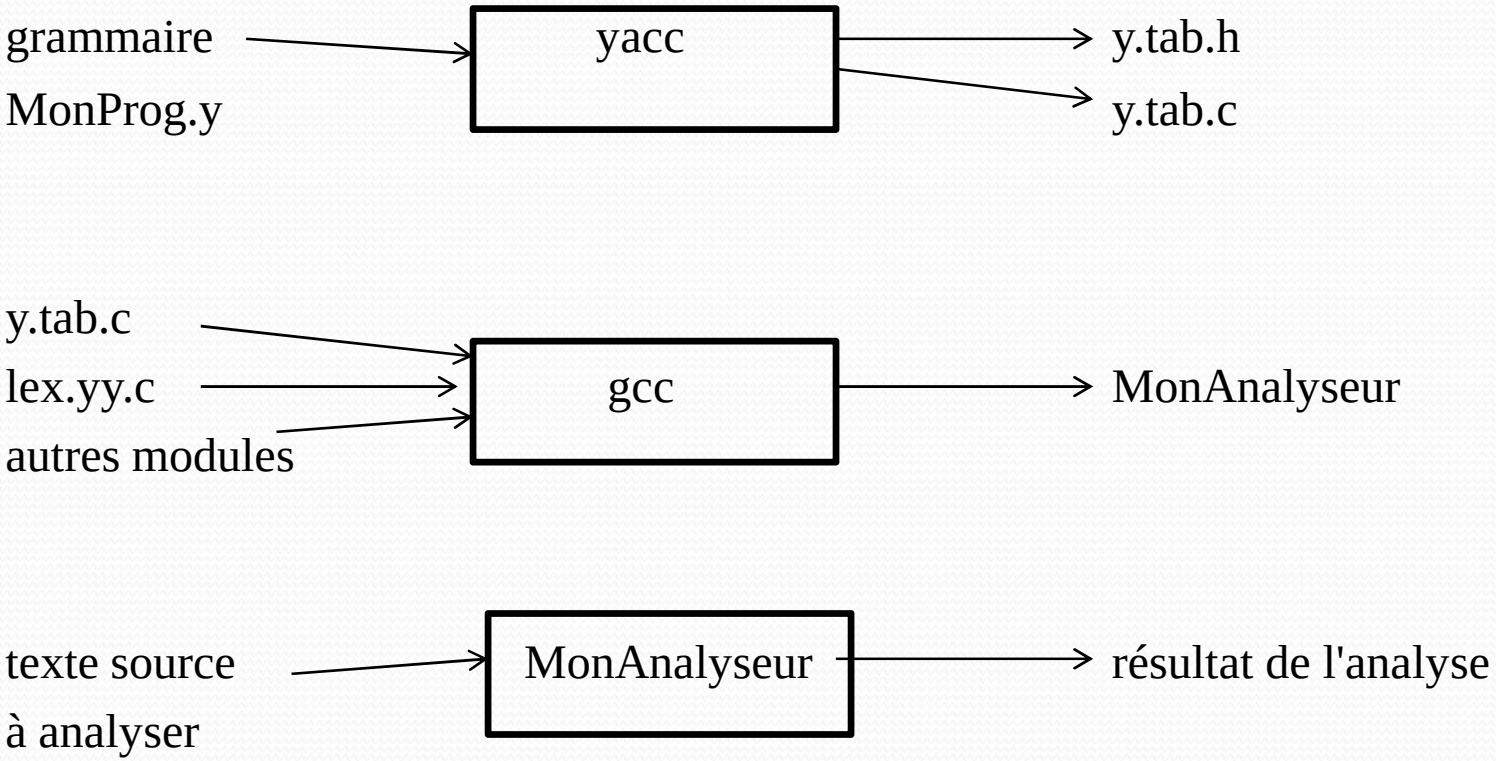
1- <u>But</u>:

Construire à partir d'une grammaire une fonction C nommée ***yyparse()***, qui est un analyseur syntaxique qui reconnait les constructions du langage décrit par la grammaire.

Un programme écrit en langage ***Yacc*** prend en entrée un fichier source constitué essentiellement des productions d'une grammaire ***G*** et produit un ***programme C*** qui, une fois compilé, est un analyseur syntaxique pour le langage ***L(G)***.

Le langage YACC

Utilisation du Langage Yacc



Le langage YACC

2- **Structure d'un code en YACC:**

Le générateur d'analyseurs syntaxiques YACC a une structure analogue à celle du LEX.

Le code source Yacc, doit avoir un nom terminé par "**.y**". Il est composé de 3 sections délimitées par deux lignes "**%%**".

```
déclarations
%%
règles de traduction
%%
routines annexes en langage C
```

Le langage YACC

2.1- **La section des déclarations**:

Elle est constituée de 2 parties:

1- Une partie entre **%{** et **%}** des déclarations à la C, des variables, unions, structures, ...

2- Une partie constituée de la déclaration de

- l'axiome, avec **%start**
- des terminaux, avec **%token**
- des opérateurs, en précisant leur associativité, avec **%left**, **%right**, **%nonassoc**

Remarques:

- *Le symbole* **%** *doit être à la 1ère colonne*
- *Tout symbole non déclaré dans cette partie par* ***token*** *est considéré comme un* ***non terminal***

Le langage YACC

2.2- La section des règles de traduction:

Après **%%** nous avons dans cette partie une suite de règles de traduction.

Chaque règle est constituée par une production de la grammaire associé aux phrases du langage à analyser et éventuellement une action sous forme d'une règle sémantique.

La règle sémantique peut prendre la forme d'un schéma de traduction ou d'une définition dirigée par la syntaxe.

Le langage YACC

2.3- La section des fonctions annexes en C:

Après **%%** nous avons dans cette partie une suite de fonctions écrite en langage C et utilisables par la 2ème partie des règles de traduction.

Remarques : la 1ère et la 3ème parties sont facultatives.

Nous allons écrire un code complet en Lex et Yacc, d'une calculatrice scientifique, qui va lire une expression arithmétique, l'évaluer et afficher le résultat.

Soit la grammaire:

R ⟶ E fin

E ⟶ E + T | E - T | T

T ⟶ T * F | T / F | F

F ⟶ (E) | nombre

Le langage YACC

Un programme en Yacc: *calc.y*

1ère partie

```
%{
#include<stdio.h>
%}
%token nombre fin // nombre et fin dont des terminaux
%left plus moins  // '+' et le '-' associatif à gauche
%left fois div
%start R // R est l'axiome
%%
```

Le langage YACC

Un programme en Yacc (suite):

2 ème partie *sans les règles sémantiques*

```
R  : E fin
   ;
E : E plus T
   | E moins T
   | T
   ;
T : T fois F
   | T div F
   | F
   ;
F : '(' E ')'
   | nombre
   ;
%%
```

Le langage YACC

2 ème partie *avec les règles sémantiques*

```
R   : E  fin            { printf("le résultat est: %d", $1);}
    ;
E : E plus T            { $$ = $1 + $3;}
    | E moins T         { $$ = $1 - $3;}
    | T                 { $$ = $1; // facultatif }
    ;
T : T fois F            { $$ = $1 * $3;}
    | T div F           { if ($3==0) printf ("division par zéro
                        interdite) else $$ = $1 / $3;}
    | F                 { $$ = $1;}
    ;
F : '(' E ')'           { $$ = $2;}
    | nombre            { $$ = $1;}
    ;
%%
```

Le langage YACC

Un programme en Yacc (suite):

3 ème partie:

```
%%
void yyerror(char *message) {
    printf("<< %s", message);
}
int main(void) {
        printf("début de l'analyse\n");
        yyparse();
        printf("fin de l'analyse\n");
}
```

L'analyseur syntaxique se présente comme une fonction `int yyparse(void)` qui rend 0 si la chaîne est acceptée, non nulle dans le cas contraire.

Le langage YACC

Un programme en Lex: *analex.l*

```
%{
#include<stdlib.h>
%include<calc.tab.h>
%}
nombre [0-9]+
%%
[ \t]+      {/* ne rien faire */}
{nombre}    {yylval = atoi(yytext); return(nombre);
"\n"        {return (fin);}
"+"         {return (plus);}
"-"         {return (moins);}
"/"         {return (div);}
"*"         {return (fois);}
%%
```

Le langage YACC

Les étapes de compilation:

1 étape: > ***bison -d calc.y***

en sortie on a, ***calc.tab.c*** et ***calc.tab.h***

2ème étape: > ***flex analex.l***

en sortie on a: ***lex.yy.c***

3ème étape: > ***gcc -c lex.yy.c -o cal.l.o***

gcc -c calc.tab.c -o calc.y.o

gcc -o calc calc.l.o calc.y.o -lfl -lm

lfl: ***Library Fast Lex*** *(la librairie du Flex)*

lm: la librairie de *Bison*

Le langage YACC

Remarque:

- L'analyseur lexical produit par *lex* transmet les attributs des unités lexicales à l'analyseur syntaxique produit par *yacc* via une variable *yylval* qui par défaut est de type `int`.

- Si nous voulons manipuler des réels, nous devons modifier ce type dans le fichier *calc.tab.h*:

```
#define YYTYPE int
...
extern YYSTYPE yylval;
```

A remplacer `int` par `double`

Le langage YACC

3- Gestion des conflits par Yacc:

Le langage Yacc adopte une *analyse ascendante par décalage-réduction*, il peut rencontrer lors de l'analyse par une grammaire 2 types de conflits:

1- Conflit entre un décalage et une réduction *(shift/reduce conflict)*, ce type correspond à choisir entre 2 actions possibles.

Le Yacc favorise par défaut le décalage.

2- Conflit entre une réduction et une autre réduction *(reduce/ reduce conflict)*, ce conflit se produit quand 2 membres droits peuvent se réduire par le symbole gauche d'une production au sommet de la pile.

Le Yacc choisit la 1ère production écrite dans le fichier Yacc.