Compilation

3- Analyse Syntaxique

L'analyse syntaxique

- Pour décrire la syntaxe d'un langage de programmation, on utilise une grammaire
- Une grammaire est un ensemble de règles décrivant comment former des phrases
- Alors que l'analyse lexicale reconnaît les mots du langage, l'analyse syntaxique en reconnaît les phrases
- Elle permet de dire si le texte source appartient au langage

Grammaires (hors contexte)

- Exemple :

  Une grammaire avec les règles de production suivantes

  1. Expr → Expr Op **NOMBRE**
  2. Expr → **NOMBRE**
  3. Op → +
  4. Op → −
  5. Op → ×
  6. Op → ÷

Grammaires (hors contexte)

Une grammaire (hors contexte) est un quadruplet G = (T, N, S0, P)

– T est l'ensemble des symboles terminaux du langage. Ils correspondent aux unités lexicales découvertes par l'analyseur lexical

– N est l'ensemble des symboles non-terminaux du langage.

– S0 ∈ N est appelé l'élément de départ de G (ou axiome de G). Le langage que G décrit (noté L(G)) correspond à l'ensemble des phrases qui peuvent être dérivées à partir de S0 par les règles de la grammaire

– P est un ensemble de productions (ou règles de réecriture) de la forme $\beta \rightarrow \alpha 1\ \alpha 2 \ldots \alpha n$ avec $\alpha i \in T \cup N$ et $\beta \in N$

“se dérive en”

- Soit αi des suites de symboles d'une grammaire.

  Si $\alpha 0 \rightarrow \alpha 1 \rightarrow \cdots \rightarrow \alpha n$ on dit que α0 **se dérive en** αn en n étapes, et on écrit :

  $\alpha 0 \rightarrow^{n} \alpha n$

- Si α **se dérive en** β en un nombre quelconque d’étapes, on écrit : $\alpha \rightarrow^{*} \beta$

Langage engendré par une grammaire

- Soit G = {N , T , S0, P } une grammaire; le langage **engendré** par G est l’ensemble des chaînes de symboles terminaux qui dérivent de S0 :

$$L(G) = \{w \in T^{*} \mid S0 \rightarrow^{*} w\}$$

Si $w \in L(G)$ on dit que w est **une phrase** de G

- Deux grammaires sont dites **équivalentes** si elles engendrent le même langage

Exemple 1

L1 défini par la grammaire suivante

1. $Z \rightarrow aXc$
2. $Z \rightarrow bY$
3. $X \rightarrow bX$
4. $X \rightarrow d$
5. $Y \rightarrow e$
6. $Y \rightarrow bY$
7. $Y \rightarrow cYX$

Est-ce que abbbdc appartient a L1 ?

Est-ce que bbae appartient a L1 ?

Est-ce que be appartient a L1 ?

Exemple 2

L2 défini par la grammaire suivante

1. $Z \rightarrow aX$
2. $X \rightarrow bY$
3. $X \rightarrow bT$
4. $Y \rightarrow c$
5. $T \rightarrow d$

Est-ce que abd appartient a L2 ?

Exemple 3

1. Z → S #
2. S → XaY
3. X → W
4. X → T
5. W → bc
6. T → ac
7. Y → eY
8. Y → f
9. Y → ε

Est-ce que aceef# appartient a L3 ?

Arbre de dérivation d'une phrase

Soit w une phrase du langage L(G) ; il existe donc une dérivation telle que S0 →*w. Cette dérivation peut être représentée par un arbre de dérivation :

- La racine de l'arbre est S0,
- Les nœuds non feuilles sont des symboles non terminaux,
- Les feuilles sont des symboles terminaux,
- Les feuilles lues de la gauche vers la droite constituent la phrase w
- Les fils d'un noeud non feuille donnent les règles utilisées :
  - Si le nœud est le symbole S et si la production

    S → S1 S2 . . . Sk a été utilisée pour dériver S alors les fils du noeud de la gauche vers la droite sont S1 S2 . . . Sk

Arbre de dérivation d'une phrase

- Exemple :

1. Expr → Expr Op **nombre**
2. Expr → **nombre**
3. Op → +
4. Op → −
5. Op → ×
6. Op → ÷

Question :

Donner l'arbre de dérivation de la phrase :

nombre-nombre×nombre ?

Arbre de dérivation d'une phrase

- Les règles appliquées pour dériver la phrase sont 1, 5, 1, 4, 2 dans l'ordre (Dérivation à droite)
- Autre séquence de règles possible : 1, 1, 2, 4, 5 (Dérivation à gauche)
- Donner l'arbre de dérivation dans chaque cas.
- Les deux arbres sont-ils identiques ?

Dans la suite ...

Soit G une grammaire et ω phrase

- Analyse syntaxique = déterminer si $\omega \in L(G)$?

On cherche d'abord une technique d'analyse descendante :

- Produire ω à partir de S0
- Produire une dérivation gauche $S0 \rightarrow^* \omega$

Et en plus on cherche à caractériser :

- Une classe de grammaires dite **LL(1)** pour laquelle une analyse efficace est possible :
  - algorithme linéaire en fonction de $|\omega|$ (d'où le premier L)
  - accès séquentiel aux symboles de ω, de gauche à droite (d'où le second L)

Premiers, Suivants

Soit G = (T, N, S0, P) une grammaire hors-contexte

Soit $\alpha \in (T \cup N)^*$, le prédicat **Vide(α)** vaut vrai ssi ε peut être dérivé de α par G :

$$\boldsymbol{\alpha \rightarrow^* \varepsilon}$$

Soit $\alpha, \beta \in (T \cup N)^*$, **Premiers(α)** est l'ensemble des symboles terminaux qui peuvent débuter une dérivation de α par G :

$$\text{Premiers}(\alpha) = \{\mathbf{a} \in \mathbf{T} \mid \boldsymbol{\alpha \rightarrow^* a\beta}\}$$

Soit $X \in N$, **Suivants(X)** est l'ensemble des symboles terminaux qui peuvent suivre une occurrence de X dans une dérivation de S0 par G :

$$\text{Suivants}(X) = \{\mathbf{a} \in \mathbf{T} \mid \mathbf{S0 \rightarrow^* \alpha X a \beta}\}$$

Directeurs

Soit $X \rightarrow u1.u2 \ldots un$ une règle de P, avec $ui \in (T \cup N)$.

- Directeurs($X \rightarrow u1.u2 \ldots un$) est l'ensemble des symboles terminaux qui peuvent "débuter" une dérivation de X par cette règle.

Directeurs($X \rightarrow u1.u2 \ldots un$) =

$(\cup_{i \in Vi}$ Premiers(ui))

$\cup$ (si Vide(u1 . . . un) alors Suivants(X) sinon ∅)

avec $Vi = \{i \mid \text{Vide}(u1 \ldots ui-1)\}$.

Grammaire LL(1)

- Une grammaire G = (T , N , S0, P) est dite **LL(1)** si et seulement si l'ensemble des règles de P définissant un même symbole non-terminal ont des directeurs disjoints :

$$\forall X \in N \,.\, \forall X \rightarrow \alpha, X \rightarrow \beta \in P$$
$$\textbf{Directeur}(X \rightarrow \alpha) \cap \textbf{Directeur}(X \rightarrow \beta) = \varnothing$$

D'où le (1) dans le nom LL(1)

- Un langage est dit LL(1) si et seulement si il existe une grammaire LL(1) qui le reconnait

Exemple

$Z \rightarrow S\#$

$S \rightarrow Xa$

$S \rightarrow \varepsilon$

$X \rightarrow bX$

Questions :

- Calculer Directeurs ($Z \rightarrow S\#$)
- La grammaire est-elle LL(1) ?

Exemple

Premiers (S) = Premiers (X) = {b}

Vide (S) = vrai

Suivants (S) = {#}

Suivant (X) = Suivants (X) $\cup$ {a} = {a}

Directeurs ($Z \rightarrow S\ \#$) = Premiers (S) $\cup$ {#} (car Vide (S))

Directeurs ($S \rightarrow Xa$) = Premiers (X) = {b}

Directeurs ($S \rightarrow \varepsilon$) = Suivants (S) = {#}

Rendre une grammaire LL(1) ?

Etant donnée une grammaire, construire G ′ telle que :

L(G ′) = L(G)

G ′ est LL(1)

- Problème : ce n'est pas toujours possible !

(langages LL(1) $\subset$ langages hors-contexte)

- Savoir si cela est possible est indécidable ...
- On propose donc des "heuristiques" pour construire G′,

Note : il n'est pas certain que la grammaire obtenue soit LL(1) . . .

1 - Elimination de la récursivité à gauche

- Une grammaire est récursive à gauche s'il existe un non-terminal A et une dérivation de la forme $A \rightarrow A\alpha$, où α est une chaîne quelconque.
- Exemple :

  La grammaire du langage d'expressions arithmétiques est récursive à gauche.
- Pour obtenir une grammaire non récursive à gauche équivalente on remplace la règle $A \rightarrow A\alpha \mid \beta$ par :

$$A \rightarrow \beta\, A' \text{ et } A' \rightarrow \alpha A' \mid \varepsilon$$

2 - Factorisation à gauche

- Soit une grammaire contenant des productions comme :

  $A \rightarrow \alpha\beta1 \mid \alpha\beta2$

- L'analyseur ne peut pas choisir entre les deux productions en se basant sur le symbole courant α

  => Pour enlever ce défaut, la grammaire peut être factorisée en transformant ces productions en :

  $A \rightarrow \alpha A'$

  $A' \rightarrow \beta1 \mid \beta2$

Exercice

- Soit la grammaire

  $Z \rightarrow I\ \#$

  $I \rightarrow i \mid i(I)$

  Est-elle LL(1) ?

  Si non donner une grammaire équivalente LL(1)

Construire un analyseur descendant

- Il faut une grammaire LL(1)
- le programme de l'analyseur est étroitement lié à la grammaire analysée :

1- Chaque groupe de productions ayant le même membre gauche S donne lieu à une procédure reconnaîtreS

2- Lorsque plusieurs productions ont le même membre gauche, le corps de la procédure correspondante est une conditionnelle qui, d'après le symbole terminal courant, sélectionne l'exécution des actions correspondant au membre droit de la production pertinente

3- Une séquence de symboles S1 S2 . . . Sn dans le membre droit d'une production donne lieu, dans la procédure correspondante, à une séquence d'instructions traduisant les actions "reconnaissance de S1", " reconnaissance de S2" □, . . .□ "reconnaissance de Sn"

Construire un analyseur descendant

4- Si S est un symbole non terminal, l'action "reconnaissance de S" se réduit à l'appel de procédure reconnaitreS.

5- Si α est un symbole terminal, l'action "reconnaissance de α" □ consiste à considérer le symbole courant et

– s'il est égal à α, passer le symbole courant au symbole suivant

– sinon, annoncer une erreur

- Remarque : on lance l'analyse en appellant la procédure associée au symbole de départ de la grammaire

Exemple

expression → terme fin_expression

fin_expression → "+" terme fin_expression | ε

terme → facteur fin_terme

fin_terme → "*" facteur fin_terme | ε

facteur → nombre | identificateur | "(" expression ")"

Donner les procédures de l'analyseur syntaxique descendant.


```
void expression() {
    terme();
    fin_expression();
}
void fin_expression() {
   if (uniteCourante == '+') {
      uniteCourante = getUnite();
      terme();
      fin_expression();
    } else erreur ...
}
void terme() {
    facteur();
    fin_terme();
}
void fin_terme() {
   if (uniteCourante == '*') {
      uniteCourante = getUnite();
      facteur();
      fin_terme();
   } else erreur ...
}
```

```
void facteur() {
if (uniteCourante == NOMBRE)
   uniteCourante = getUnite();
else if (uniteCourante == IDF)
   uniteCourante = getUnite();
else {
   if (uniteCourante == '(')
     uniteCourante = getUnite();
   else erreur ...
   expression();
   if (uniteCourante == ')')
     uniteCourante = getUnite();
   else erreur ...
}
```


Exercice

1. S → aSc
2. S → R
3. R → bRc
4. R → ϵ

Existe-t-il un analyseur descendant LL(1) pour cette grammaire ?

Si oui donner les procédures associées.

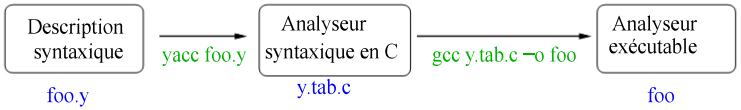

```
void reconnaitre_S() {
 switch (uniteCourante){
   if (uniteCourante == XX)    code règle 1
   else if (uniteCourante == YY)   code règle 2
   else erreur("XX ou YY attendu");
 }
}
```

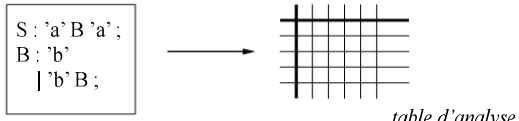
Que doivent être XX et YY ?

Analyseur descendant : bilan

1. Passer la grammaire en LL(1) si on y arrive
2. Calculer les directeurs
3. Ecrire une procédure par non-terminal qui :
   - teste le premier lexème
   - choisit la règle à appliquer

Fonctionnement de Yacc





liste de productions

table d'analyse

Organisation de la description syntaxique

Déclarations pour le programme C
```
%{
   int i;
%}
```
Déclaration de propriétés de symboles pour Yacc
```
%start S

%%
```
Règles de production de la grammaire et actions sémantiques
```
S : 'a' B 'a'       { printf("..."); }
;
B : 'b'             { printf("..."); }
   | 'b' B          { printf("..."); }
;
%%
```
Fonctions et main C
```
int main () { ... }
```

Déclaration de propriétés de symboles

```
%union {
    déclaration C d'un champ d'union
}
```
Terminaux

%token<nom de champ> liste de terminaux

Non-terminaux

%type<nom de champ> liste de non-terminaux

Associativité des non-terminaux

%left liste de terminaux
%right liste de terminaux

Racine

%start non-terminal

Déclarations pour Yacc

- Déclarations des unités lexicales. Ex :

%token NOMBRE IDENTIF

%token EGAL DIFF INFEG SUPEG

%token SI ALORS SINON

- Ces déclarations d'unités lexicales intéressent
  - Yacc, qui les utilise,
  - mais aussi lex, qui les manipule en tant que résultats de la fonction yylex.
- Pour cette raison, yacc produit un fichier y.tab.h, destiné à être inclus dans le source lex

y.tab.h

pour les déclarations ci-dessus y.tab.h contient :

#define NOMBRE 257

#define IDENTIF 258

#define EGAL 259

...

#define ALORS 264

#define SINON 265

Symboles dans Yacc

- Les caractères simples, encadrés par des apostrophes sont tenus pour des **symboles terminaux,**
- Les identificateurs mentionnés dans les déclarations %token sont tenus pour des **symboles terminaux**,
- Tous les autres identificateurs apparaissant dans les productions sont considérés comme des **symboles non terminaux**,
- Par défaut, **le symbole de départ** est le membre gauche de la première règle Yac.

Règles de production

- Le méta-symbole → est indiqué par deux points
- Chaque groupe de règles ayant même membre gauche est terminée par un point-virgule;
- La barre verticale | a la même signification que dans la notation des grammaires.

Grammaire (G1) :

expr → expre "+" terme | terme
terme → terme "*" facteur | facteur
facteur → nombre | identif | "(" expr ")"

(G1) en Yacc :

```
%token nombre identif
%%
expr : expr '+' terme | terme ;
terme : terme '*' facteur | facteur ;
facteur : nombre | identif | '(' expr ')' ;
```

Fonction d'analyse et fonction d'erreur

- Focntion d'analyse : int yyparse(void) :
  - Rend 0 lorsque la chaîne d'entrée est acceptée
  - Rend une valeur non nulle dans le cas contraire.

Exemple code Yacc :

```
%%
int main(void) {
    if (yyparse() == 0)
        printf("Phrase correcte\n");
}
```

- Fonction d'erreur : int yyerror(char * message)

Exemple code Yacc :

```
%%
void yyerror(char *message) {
    printf(" <<< %s\n", message);
}
```

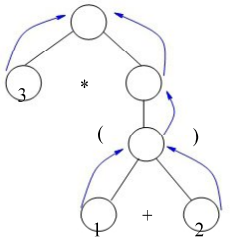
Actions sémantiques

En général: Commandes C comprises entre { ... }

```
B : 'b'          { printf("règle B1"); }
  | 'b' B        { printf("règle B2"); }
;
```

Accès aux sous-arbres:

```
E : E '+' E      { $$ = $1 + $3; }
  |  ....
  | '(' E ')'    { $$ = $2; }
;
```



Actions sémantiques

- Une action sémantique est une séquence d'instructions C écrite, entre accolades, à droite d'une production.
- Cette séquence est recopiée par yacc dans l'analyseur produit, de telle manière qu'elle sera exécutée, pendant l'analyse, lorsque la production correspondante aura été employée.
- Exemple :

```
facteur : nombre        { printf(" %d", yylval); }
        | identif       { printf(" %s", nom); }
        | '(' expr ')'
        ;
```

Exemple : arith.l

```
%{
#include "syntaxe.tab.h"
extern char nom[]; /* chaîne de caractères partagée avec l'analyseur syntaxique */
%}
chiffre [0-9]
lettre [A-Za-z]
%%
[" "\t\n]                   { }
{chiffre}+                  { yylval = atoi(yytext); return nombre; }
{lettre}({lettre}|{chiffre})*  { strcpy(nom, yytext); return identif; }
.                           { return yytext[0]; }
%%
int yywrap(void) {
        return 1;
}
```

Exemple : arith.y

```
%{
char nom[256]; /* chaîne de caractères partagée avec l'analyseur lexical */
%}
%token nombre identificateur
%%
expression : expression '+' terme { printf(" +"); }
           | terme
           ;
terme : terme '*' facteur { printf(" *"); }
      | facteur
      ;
facteur : nombre { printf(" %d", yylval); }
        | identificateur { printf(" %s", nom); }
        | '(' expression ')'
        ;
```

Exemple : arith.y (suite)

```
%%
void yyerror(char *s) {
        printf("<<< \n%s", s);
}
main() {
       if (yyparse() == 0)
              printf(" Expression correcte\n");
}
```

Attribut d'un symbole

- **Attribut** : Valeur que possède un symbole, terminal ou non terminal.
- Exemple :
  - Reconnaissance du lexème "2001" donne lieu à l'unité lexicale NOMBRE =>

    Le symbole NOMBRE a pour attribt la valeur 2001.
- Un analyseur lexical produit par **lex** transmet les attributs des unités lexicales à un analyseur syntaxique produit par yacc à travers une variable :
  - Nommé **yylval**,
  - Par défaut, son type est **int**.

Ajout des attributs dans les actions sémantiques

- Pour mettre des attributs, on utilise des notations :
  - $1, $2, $3, etc. désignent les valeurs des attributs des symboles constituant le **membre droit** de la production concernée par l'action sémantique,
  - $$ désigne la valeur de l'attribut du symbole qui est le **membre gauche** de cette production.
- L'action sémantique { $$ = $1 ; } est implicite

  => Il n'y a pas besoin de l'écrire.

Exemple : Calculateur (arith-calc.y)

```
%{
    void yyerror(char *);
%}
%token nombre
%%
session     : session expr '=' { printf("résultat : %d\n", $2); }
            |
            ;
expr        : expr '+' terme { $$ = $1 + $3; }
            | expr '-' terme { $$ = $1 - $3; }
            | terme
            ;
terme       : terme '*' facteur { $$ = $1 * $3; }
            | terme '/' facteur { $$ = $1 / $3; }
            | facteur
            ;
facteur     : nombre
            | '(' expr ')' { $$ = $2; }
            ;
```

Exemple : Calculateur (suite de arith-calc.y)

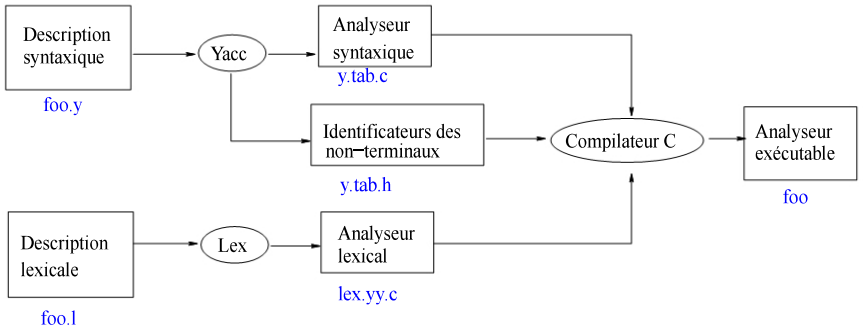
```
%%
void yyerror(char *s) {
     printf("<<< \n%s", s);
}
main() {
      yyparse();
      printf("Au revoir!\n");
}
```

Exercice :

Tester l'exemple Calculateur avec les expressions :

- 10 + 50
- 1000 – 50 - 250
- (10 + 50)*(1000 – 50 - 250)

Schéma de compilation



Commandes:

```
> yacc -d foo.y
> lex foo.l
> gcc y.tab.c lex.yy.c -ll -o foo
```

Typage des valeurs (%union)

- Quand les mots reconnus par l'analyseur lexical peuvent avoir des valeurs de types différents. Exemple :
  - un nombre entier ou
  - un nombre flottant ou
  - une chaîne de caractères,
- alors yylval sera défini comme une union en C.
- L'analyseur syntaxique utilisera le type renvoyé pour savoir quel champs de l'union il doit utiliser.

Typage des valeurs (%union)

Exemple :

- Avec la déclaration :

```
%union {
        Float f;
 }
```

on a prévenu Yacc que certains symboles **auraient une valeur du type** float.

- Il faut maintenant **spécifier le type de valeur** attaché à chaque symbole avec une déclaration.
  - Pour les symboles terminaux (ex : nombre), cela est effectué au moment de sa déclaration :

    %term <f> nombre
  - Pour les symboles non déclarés (ex: expr), il faut a jouter nouvelle déclaration :

    %type <f> expr