Traduction et Sémantique
Analyse descendante, l'outil Yacc

Luigi Santocanale
LIF, Université de Provence
Marseille, FRANCE

12 février 2010

Plan

Plan

$w \in L(\mathscr{G})$: analyse descendante récursive

- Construire un arbre de dérivation (ou une DG) de w à partir de l'axiome S.
- Essayer, de façon récursive, toutes le productions …
- …pas efficace, mais certaines fois on peut être chanceux.
- On peut s'aider en lisant un morceaux du mot w (prévision)
- La pile implicite des appels récursifs peut être substituée par une pile explicite d'un AP.

Analyse LL(1)

Idée :

- en lisant le mot w à partir de la gauche
  – left-to-right parse
- en lisant un caractère à la fois
  – lookahead 1
- reconstruire une dérivation gauche du mot w à partir du symbole S
  – leftmost derivation

Un exemple

Soit $\mathscr{G}$:

$$S \to aT$$
$$T \to bT \mid c$$

Le mot $abc \in L(\mathscr{G})$ car

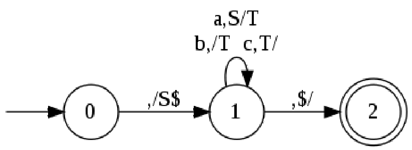
$$\begin{array}{ccccccc} abc & & xbc & & xxc & & xxx \\ S & \Rightarrow & aT & \Rightarrow & abT & \Rightarrow & abc \end{array}$$

Le mot $aba \notin L(\mathscr{G})$ car

$$\begin{array}{ccccccc} aba & & xba & & xxa & & \\ S & \Rightarrow & aT & \Rightarrow & abT & \Rightarrow & \texttt{err} : \end{array}$$

aucune règle à appliquer.

On peut implémenter cet algorithme par un APD :



On a, par exemple :

$$(0, abc,) \vdash (1, abc, S\$) \vdash (1, bc, T\$) \vdash (1, c, T\$) \vdash (1, , \$) \vdash (2, ,)$$

Vers une généralisation

- Un non-terminal A peut « engendrer à gauche » $a \in \Sigma$, par une suite de productions :

$$A \to BC$$
$$B \to aB \mid \dots$$

- Un non-terminal A peut s'effacer, c'est-à-dire engendrer ε :

$$A \to BC$$
$$B \to \varepsilon \mid \dots$$
$$C \to aD \mid \varepsilon \mid \dots$$

Si la recherche est guidée par un couple (a, A), on peut dépiler A et considérer le « symbole suivant » sur la pile.

Obstacles

On veut construire un APD. Cela n'est pas possible si :

- plusieurs productions du même non-terminal engendrent a :

$$\begin{aligned} A &\to aB \mid CD \\ C &\to aE \mid \dots \end{aligned}$$

- un non-terminal A
  - peut s'effacer,
  - engendre a,
  - un non-terminal B, qui peut suivre A dans la pile, engendre aussi a :

$$\begin{aligned} A &\to aB \mid \varepsilon \\ B &\to aD \\ E &\to AB \end{aligned}$$

NULL, FIRST et FOLLOW

Pour $\mathscr{G} = \langle V, \Sigma, S, P \rangle$, posons :

$$\begin{aligned} NULL &= \{\,\text{non-termiaux qu'on peut « effacer »}\,\} \\ &= \{\, A \in V \mid A \Rightarrow^* \varepsilon \,\} \end{aligned}$$

$$\begin{aligned} FIRST(X) &= \{\,\text{terminaux qu'on peut « engendrer à gauche » à partir de } X \in V\,\} \\ &= \{\, a \in \Sigma \mid X \Rightarrow^* a\alpha \,\} \end{aligned}$$

$$\begin{aligned} FOLLOW(A) &= \{\,\text{terminaux peuvant suivre } A \text{ dans un premier morceaux d'une dérivation}\,\} \\ &= \{\, b \in \Sigma \mid S \Rightarrow^* \alpha A b \beta \,\} \end{aligned}$$

Le calcul de NULL

NULL est le plus petit ensemble sous-ensemble de V tel que :

1. si $A \to \varepsilon$, alors $A \in NULL$,
2. si $A \to X_1 \dots X_n$ et $X_1, \dots, X_n \in NULL$, alors $A \in NULL$.

Un algorithme standard :

```
null= ∅
null_next = {A ∈ V | A → ε}
tant que null != null_next
faire
  null=null_next
  pour toute production A → X_1 ... X_n
     si X_1,...,X_n ∈ null
         ajouter A à null_next
  fin pour
fin faire
retourner null
```


Le calcul de FIRST

$\{\, FIRST(X) \mid X \in V \,\}$
est la plus petite collection de sous-ensembles de Σ telle que :

1. si $X = a \in \Sigma$, alors $FIRST(X) = \{\, a \,\}$,
2. si

$$X \to Y_1 \dots Y_n Z \alpha$$

et $Y_1, \dots, Y_n \in NULL$, alors $FIRST(Z) \subseteq FIRST(X)$.

En fait, si $a \in FIRST(Z)$:

$$X \Rightarrow Y_1 \dots Y_n Z\alpha \Rightarrow \dots \Rightarrow Z\alpha \Rightarrow \dots \Rightarrow a\alpha$$

Caractérisation de FOLLOW (I)

Si $c \in FIRST(X)$, alors :

$$S \Rightarrow \ldots \Rightarrow \ldots A \ldots \Rightarrow \ldots \alpha B Y_1 \ldots Y_n X \beta \ldots \Rightarrow \ldots$$
$$\Rightarrow \ldots \alpha B X \beta \ldots \Rightarrow \ldots \Rightarrow \ldots \alpha B c \beta \ldots$$

Si $S \Rightarrow^* \ldots A d \ldots$ – c'est-à-dire $d \in FOLLOW(A)$:

$$S \Rightarrow \ldots \Rightarrow \ldots A \ldots \Rightarrow \ldots \alpha B Y_1 \ldots Y_n \ldots \Rightarrow \ldots$$
$$\Rightarrow \ldots \alpha B \ldots \Rightarrow \ldots \Rightarrow \ldots \alpha B d \ldots$$

Caractérisation (et calcul) de FOLLOW (II)

$\{ FOLLOW(B) \mid B \in V \setminus \Sigma \}$
est la plus petite collection de sous-ensembles de Σ telle que :

1. si
$$A \to \alpha B Y_1 \ldots Y_n X \beta$$
et $Y_1, \ldots Y_n \in NULL$, alors $FIRST(X) \subseteq FOLLOW(B)$,
2. si
$$A \to \alpha B Y_1 \ldots Y_n$$
et $Y_1, \ldots Y_n \in NULL$, alors $FOLLOW(A) \subseteq FOLLOW(B)$.

Pour le caluler, on utilise un algorithme standard pour chercher la plus petite solution de contraintes monotones.

Un exemple : FIRST

Soit $\mathscr{G}$:

$$E \to T\ E' \qquad E' \to +\ T\ E' \mid \varepsilon$$
$$T \to F\ T' \qquad T' \to *\ F\ T' \mid \varepsilon$$
$$F \to (\ E\ ) \mid id$$

Les contraintes :

$$FIRST_E \supseteq FIRST_T \qquad FIRST_{E'} \supseteq \{+\}$$
$$FIRST_T \supseteq FIRST_F \qquad FIRST_{T'} \supseteq \{*\}$$
$$FIRST_F \supseteq \{(, id\}$$

La solution :

$$FIRST_E = \{(, id\} \qquad FIRST_{E'} = \{+\}$$
$$FIRST_T = \{(, id\} \qquad FIRST_{T'} = \{*\}$$
$$FIRST_F = \{(, id\}$$

Un exemple : FOLLOW

Soit $\mathscr{G}$:

$$E \to T\ E' \qquad E' \to +\ T\ E' \mid \varepsilon$$
$$T \to F\ T' \qquad T' \to *\ F\ T' \mid \varepsilon$$
$$F \to (\ E\ ) \mid id$$

Les contraintes :

$$FOLLOW_E \supseteq \{)\} \qquad FOLLOW_{E'} \supseteq FOLLOW_E$$
$$FOLLOW_T \supseteq \{+\} \cup FOLLOW_E \qquad FOLLOW_{T'} \supseteq FOLLOW_T$$
$$FOLLOW_F \supseteq \{*\} \cup FOLLOW_T$$

La solution :

$$FOLLOW_E = \{)\} \qquad FOLLOW_{E'} = \{)\}$$
$$FOLLOW_T = \{+,)\} \qquad FOLLOW_{T'} = \{+,)\}$$
$$FOLLOW_F = \{*, +,)\}$$

Grammaires LL(1)

Pour $\gamma \in V^*$ soit

$$FIRST(\gamma) = \{ a \in \Sigma \mid \gamma \Rightarrow^* a\alpha \}$$

de façon que :

$$FIRST(X_1X_2 \dots X_n) = \bigcup_{i=1,\dots,n} \{ FIRST(X_i) \mid X_1, \dots, X_{i-1} \in NULL \}.$$

Définition : Une grammaire $\mathscr{G}$ est $LL(1)$ si, pour tout A tel que

$$A \to \alpha_1 \mid \dots \mid \alpha_n$$

on a :

- $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \emptyset$ si $i \neq j$,
- si $A \in NULL$, alors $FIRST(\alpha_i) \cap FOLLOW(A) = \emptyset$, pour $i = 1, \dots, n$.

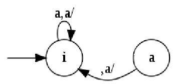
Construction de l'APD

- Etant donné $\mathscr{G}$, on construit un AP $M_{\mathscr{G}} = \langle \Sigma, Q, i, \Gamma, \Delta \rangle$, tel que :
  - $Q = \{ i \} \cup \{ a \mid a \in \Sigma \}$,
  - $\Gamma = V$,
  - Δ : page suivante.
- $M_{\mathscr{G}}$ est une APD si $\mathscr{G}$ est $LL(1)$.
- Cet AP accepte par ruban et pile vide.
- Le AP
  1. lit un caractère a et le sommet de la pile A,
  2. si $A \to \alpha$ avec $a \in FIRST(\alpha)$, alors construit l'étape
  $$wA\beta \Rightarrow w\alpha\beta$$
  d'une dérivation gauche,
  3. si $A \Rightarrow^* \varepsilon$ avec $b \in FOLLOW(A)$, alors construit les étapes
  $$wA\beta \Rightarrow^* w\beta$$
  d'une dérivation gauche.

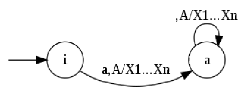
Les transitions

Δ est de la forme :

1. Consommer a :

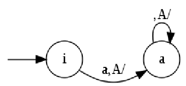


2. Suivre une production :



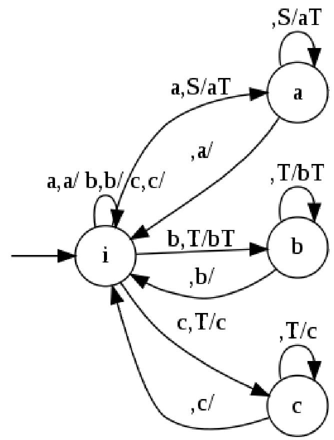
si $A \to X_1 \dots X_n$ et $a \in FIRST(X_1 \dots X_n)$.

3. Effacer un non-terminal :



où $A \in NULL$ et $a \in FOLLOW(A)$.

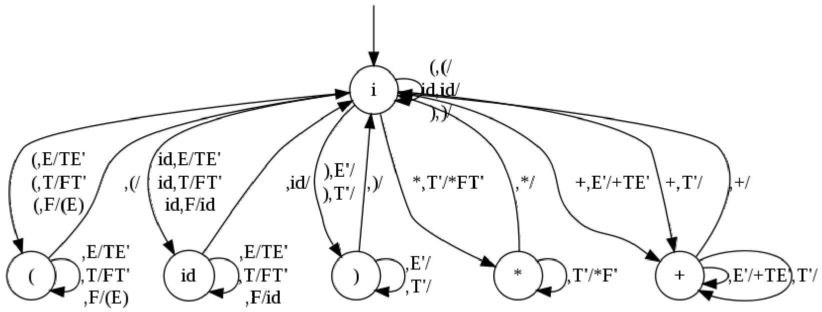
Notre premier exemple



… et un calcul :

$$\begin{aligned}(i, abc, S) &\vdash (a, bc, aT)\\ &\vdash (i, bc, T)\\ &\vdash (b, c, bT)\\ &\vdash (i, c, T)\\ &\vdash (c, , c)\\ &\vdash (i, ,)\end{aligned}$$

Le deuxième exemple



Plan

Analyse et grammaires LL(1)
- Des idées
- NULL, FIRST et FOLLOW
- Construction de l'APD

Les outils yacc et bison
- Les débuts
- Ajout d'actions

Histoire de Yacc

- Écrit entre 1975 et 1978 par S.C. Johnson at Bell Labs
- « Yet Another Compiler Compiler »
- Berkeley yacc, écrit en 1985 par B. Corbett
  algorithmes améliorés, licence Berkeley
- Bison : évolution du Bekely Yacc,
  distribué sous licence GNU (GPL)

Structure d'un fichier yacc : déclarations

```
/** Déclarations **/

%{
/* Décl. du code C */
#include <stdio.h>
#include <stdlib.h>
%}

/* Décl. des terminaux */
%token PLUS MULT
%token PG PD
%token ID
```

Structure d'un fichier yacc : grammaire

```
%%
 /** Section Grammaire **/

exp: term expp
;
expp: PLUS term expp
      |
;
term: facteur termp
;
termp: MULT facteur termp
       |
;
facteur: PG exp PD
       | ID
;

%%
```

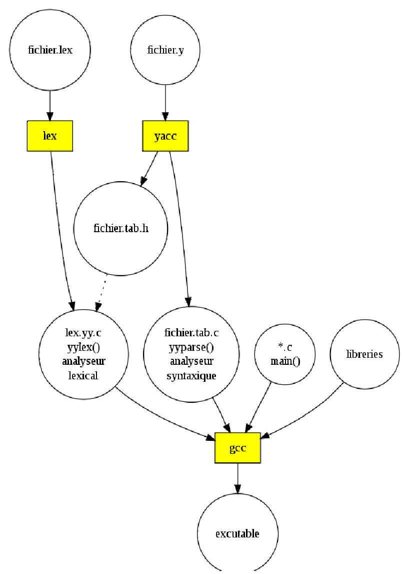

Structure d'un fichier yacc : code

```
/** Section Code C **/
int
yyerror(char *str){
  fprintf(stderr,"erreur : %s\n",str);
  exit(EXIT_FAILURE);
}

int
main(int argc, char * argv[]){
  yyparse();
  printf("Analyse terminee\n.");
  exit(EXIT_SUCCESS);
}
```


Intéractions avec lex et le lexeur

À la compilation :



Le fichier lex et son include

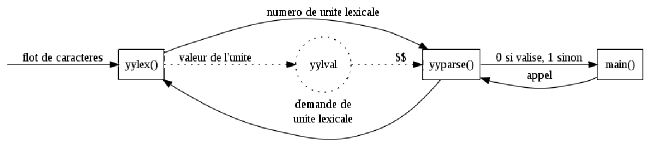
Le fichier lex :

```
%{
#include "yaccexemple.tab.h"
%}
%option noyywrap
%%
[a-zA-Z]+ {return ID;}
\( {return PG;}
\) {return PD;}
\+ {return PLUS;}
\* {return MULT;}
%%
```

Le fichier `yaccexemple.tab.h` :

```
/* Tokens.  */
#define PLUS 258
#define MULT 259
#define PG 260
#define PD 261
#define ID 262
```

À l'exécution



Structure d'un fichier yacc : déclarations

```
/** Déclarations **/

%{
/* Décl. du code C */
#include <stdio.h>
#include <stdlib.h>
%}

/* Décl. des terminaux */
%token PLUS MULT
%token PG PD
%token CONST
```


Structure d'un fichier yacc : grammaire

```
%%
 /** Section Grammaire **/
start: exp {printf("Resultat : %d\n",$1);}
;
exp: term expp    {$$ = $1 + $2;}
;
expp: PLUS term expp {$$ = $2 + $3;}
     | {$$ = 0;}
;
term: facteur termp  {$$ = $1 * $2;}
;
termp: MULT facteur termp {$$ = $2 * $3;}
      | {$$ = 1;}
;
facteur: PG exp PD {$$=$2;}
| CONST {$$=$1;}
;

%%
```


Structure d'un fichier yacc : code

```
/** Section Code C **/
int
yyerror(char *str){
  fprintf(stderr,"erreur : %s\n",str);
  exit(EXIT_FAILURE);
}

int
main(int argc, char * argv[]){
  yyparse();
  printf("Analyse terminee.\n");
  exit(EXIT_SUCCESS);
}
```

Le fichier lex et son include

Le fichier lex :

```
%{
#include "yaccexemple2.tab.h"
%}
%option noyywrap
%%
[0-9]+ {
  yylval = atoi(yytext);
  return CONST;}
\( {return PG;}
\) {return PD;}
\+ {return PLUS;}
\* {return MULT;}
%%
```

Le fichier `yaccexemple2.tab.h` :

```
/* Tokens.  */
#define PLUS 258
#define MULT 259
#define PG 260
#define PD 261
#define CONST 262
```