Analyse Syntaxique ascendante

- OcamlYacc: un générateur d'analyseurs LALR(1)
- Structure de la source OcamlYacc
- Déclarations et typage implicite
- Utiliser les grammaires ambiguës
  - Précedence
  - Associativité
  - Traiter la Conditionnelle
- Exemples importants
  - lire la sortie de OcamlYacc
  - adapter une grammaire: opérateurs
- L'exemple classique: la calculette en OcamlYacc/OcamlLex.

Yacc et OcamlYacc

La construction des tables LALR(1) pour un vrai langage produit des automates avec plusieurs centaines d'états (ma grammaire pour le langage du projet en a 125), donc il faut utiliser des générateurs automatiques qui prennent une description de la grammaire et produisent un analyseur.

Tel est le but de Yacc (ou Bison) qui produisent un analyseur écrit en C, et de OcamlYacc, qui produit un analyseur écrit en Ocaml.
Ces générateurs d'analyseurs sont fait pour travailler en tandem avec Lex (ou Flex) pour C, et OcamlLex, respectivement.

La grammaire des sommes, en OcamlYacc

La grammaire[a]

$$\begin{array}{lcl} S & \rightarrow & E\ \$ \\ E & \rightarrow & T + E \mid T \\ T & \rightarrow & id \end{array}$$

est décrite dans le fichier source OcamlYacc à coté:

[a] vue dans la partie théorique

```
%{
%}
/* un commentaire OcamlYacc! */
%token EOF ID PLUS
%start s
%type <unit> s
%%
s:  e EOF {}
;
e:     t PLUS e {}
     | t {}
;
t:     ID {}
;
%%
```


Structure de la source OcamlYacc

Comme pour OcamlLex, la source est divisée en plusieurs parties logiques:

```
%{
  déclarations et code Ocaml utilisés dans les actions
  de l'analyseur (partie optionnelle)
%}
  directives et déclarations pour OcamlYacc
%%
  description des règles de la grammaire
%%
  déclarations et code Ocaml qui utilisent les
  fonctions d'analyses produites par OcamlYacc
  (partie optionnelle)
```

Directives et déclarations pour OcamlYacc

%token ***symbol...symbol*** Déclaration des symboles terminaux.

%token < ***type*** > ***symbol...symbol*** Déclaration de tokens ayant une valeur sémantique associée de type *type*.

%start ***symbol...symbol*** Déclaration des points d'entrée.

%type < ***type*** > ***symbol...symbol*** Déclaration du type renvoyé par la fonction analysant le non terminal donné. Nécessaire seulement sur le points d'entrée (à différence de Yacc!)

Enfin, précédence et associativité (on y revient):

%left ***symbol...symbol***
%right ***symbol...symbol***
%nonassoc ***symbol...symbol***

Description des règles de la grammaire

Les lignes

```
e:     t PLUS e {}[1]
     | t {}[2]
;
```

décrivent les deux productions

$$\begin{array}{lcl} E & \rightarrow & E + T \\ E & \rightarrow & T \end{array}$$

Entre accolades on peut mettre du code Ocaml, qui a accès à des valeurs `$1`...`$n`, sur lesquels on reviendra.

[1] ici, des actions *sémantiques*
[2] ici aussi

Appel de OcamlYacc

Si `exmpl.mly` contient le source OcamlYacc, alors la commande

```
camlyacc exmpl.mly
```

produit les fichiers

exmpl.mli la description de l'interface de l'analyseur.
Cela contient aussi la définition d'un type Ocaml `token`[3] *contenant tous les tokens déclarés avec la directive* `%token`.

exmpl.ml l'analyseur syntaxique produit par OcamlYacc.
Cela définit une fonction

```
fnt: (Lexing.lexbuf -> token) -> Lexing.lexbuf -> t
```

pour chaque symbole non terminal *fnt* déclaré dans une directive %start et dont le type t est donné dans une directive %type.

Appel de OcamlYacc (suite)

Si `exmpl.mly` contient le source OcamlYacc, alors la commande

```
ocamlyacc -v exmpl.mly
```

produit en plus le fichier

exmpl.output la description de l'automate LALR(1).

Utiliser des grammaires ambiguës

OcamlYacc (comme Yacc), permet (et encourage) l'utilisation de grammaires ambiguës, pour lesquelles on utilise les directives *%left, %right et %nonassoc* pour spécifier associativité et précédence:

```
/* si on veut lire -1+3*4*5=-4+3+4
   comme ((-1)+((3*4)*5))=(((-4)+3)+4), on peut ecrire  */
                  /* precédence    associtivité  */
%nonassoc EQUAL   /*    faible         non       */
%left PLUS        /*  moyenne        à gauche    */
%left MULT        /*   élevée        à gauche    */
%nonassoc UMINUS  /* plus elevée       non       */
```

N.B.: `nonassoc` signifie que l'opérateur n'est pas associatif, et donc en particulier `3=4=5` ne peut pas être reconnu.

[3] On peut ensuite utiliser ce fichierdans un fichier d'entrée pour OcamlLex,ce qui permet d'écrire la définition des tokensune seule fois à un seul endroit.

Exemple

Donc pour la grammaire

$$S \rightarrow E\ \$ \qquad E \rightarrow E + E \mid id$$

plutôt que disambiguer[4] on peut utiliser le source OcamlYacc suivant qui est équivalent, mais plus rapide

```
%token EOF ID PLUS
%right PLUS
%start s
%type <unit> s
%%
s:     e EOF {};
e:     e PLUS e {}
     | ID {};
```

Précédences, plus en détail

Quand OcmalYacc trouve un conflit shift/reduce dans un état, et dispose d'une série de déclarations de précédences données dans le source OcamlYacc, il procède comme suit:

- il associe à la règle utilisée pour reduce la priorité pr du symbole terminal le plus à droite
- il compare cette priorité avec celle ps_t de chaque terminal t pour lequel on devrait faire un shift
  - si $pr < ps_t$ alors il choisit shift
  - si $pr > ps_t$ alors il choisit reduce
  - si $pr = ps_t$ alors on a déclaré les symboles sur la même ligne, avec la même directive
    - si ell'est `%left` on reduit
    - si ell'est `%right` on décale
    - si ell'est `%nonassoc` on met dans la case une action erreur

Précédences, plus en détail: exemple

Dans la grammaire ambigue des expressions arithmétiques avec la déclaration

```
%left PLUS
%left MULT
```

on résout le conflit

[4]
S → E $
E → T
E → T + E
T → id

```
e .  MULT e
e PLUS e .
```

par un shift, parce-que MULT a précédence supérieure à PLUS, et le conflit

```
e .  PLUS e
e PLUS e .
```

par un reduce, parce-que PLUS est déclaré associer à gauche

Précédences, directive %prec

On peut expliciter la précédence d'une règle avec des terminaux fictifs (on utilise ce terminal à la place du terminal le plus à droit):

$$S \rightarrow E\,\$ \qquad E \rightarrow id$$
$$E \rightarrow E + E \qquad E \rightarrow let\ id\ equal\ E\ in\ E$$

```
%token EOF ID PLUS LET EQUALS IN
%nonassoc LETPREC
%right PLUS
%start s
%type <unit> s
%%
s:     e EOF {};
e:     e PLUS e {}
     | ID {}
     | LET ID EQUALS e IN e   %prec LETPREC {};
```

Précédences, if then else

Un autre cas typique est

$$S \rightarrow E\,\$ \qquad E \rightarrow if\ E\ then\ E\ [\,else\ E\,]$$
$$E \rightarrow id$$

```
%token EOF IF THEN ELSE ID
%start s
%type <unit> s
%%
s:  e EOF {};
e:     ID {}
     | IF e THEN e {}
     | IF e THEN e ELSE e  {};
```

Précédences, if then else (suite)

$$S \rightarrow E\,\$ \qquad E \rightarrow if\ E\ then\ E\ [\,else\ E\,]$$
$$E \rightarrow id$$

L'ambiguïté produit le conflit:

Pièges à éviter

Il faut

```
%token EOF ID PLUS MULT MINUS
%start s
%right PLUS MINUS
%left MULT
%type <unit> s
%%
s:  e EOF {};
e:
       e PLUS e {}
     | e MULT e {}
     | e MINUS e {}
     | ID {};
```



```
10: shift/reduce conflict (shift 11, reduce 3) on ELSE
state 10
e : IF e THEN e .  (3)
e : IF e THEN e . ELSE e  (4)

ELSE  shift 11
EOF  reduce 3
THEN  reduce 3
```

OcamlYacc choisit toujours *shift* dans un conflit *shift/reduce*.
C'est bien ici, donc on ne modifie pas la source.

Pièges à éviter

Ceci est attrayant, …

```
%token EOF ID PLUS MULT MINUS
%start s
%right PLUS MINUS
%left MULT
%type <unit> s
%%
s:     e EOF {};
e:     e op e {}
     | ID {};
op:
       PLUS  {}
     | MULT  {}
     | MINUS {};
```

Pièges à éviter

… mais mauvais (et les précédences ne règlent pas le problème)

```
11: shift/reduce conflict (shift 7, reduce 2) on PLUS
11: shift/reduce conflict (shift 8, reduce 2) on MULT
11: shift/reduce conflict (shift 9, reduce 2) on MINUS
state 11
e : e . op e  (2)
e : e op e .  (2)

PLUS  shift 7
MULT  shift 8
MINUS  shift 9
EOF  reduce 2

op  goto 10
```