Analyse Syntaxique ascendante

- OcamlYacc: un générateur d'analyseurs LALR(1)
- L'exemple déjà vu
- Structure de la source OcamlYacc
- Déclarations et typage implicite
- Utiliser les grammaires ambiguës
  - Précedence
  - Associativité
  - 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 suivante, vue dans la partie théorique

$$\begin{array}{lllllll} S & \rightarrow & E\,\$ & & E & \rightarrow & T \\ E & \rightarrow & T + E & & T & \rightarrow & id \end{array}$$

est décrite dans le fichier source OcamlYacc suivant

```
%{
%}
/* ceci est un commentaire: comme en C, mais seulement pour O-
camlYacc! */
%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
%}
  directives et déclarations pour OcamlYacc
%%
  description des règles de la grammaire
%%
  déclarations et code Ocaml qui utilisent
  les fonctions d'analyses produites par OcamlYacc
```

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!)

%left ***symbol... symbol***

%right ***symbol... symbol***

%nonassoc ***symbol... symbol*** Déclaration de précédence et associativité (on les verra plus en bas).

Description des règles de la grammaire

Les lignes

```
e:      t PLUS e {}
      | t {}
;
```

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.

Appel de OcamlYacc

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

```
ocamlyacc 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 contenant tous les tokens déclarés avec la directive %token.
On peut ensuite utiliser ce fichier dans un fichier d'entrée pour OcamlLex, ce qui permet d'écrire la définition des tokens une seule fois à un seul endroit.

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:

```
/* on veut lire -1+3*4*5=-4+3+4 comme ((-1)+((3*4)*5))=(((-
4)+3)+4)  */
%nonassoc EQUAL    /* precédence faible, associtivité non spécifiée  */
%left MULT         /* précédence moyenne, associativité à gauche     */
%left PLUS         /* précédence plus élevée, associativité à gauche */
%nonassoc UMINUS   /* précédence encore plus relevée, asso-
ciativité non spécifiée */
```

Exemple

Donc pour la grammaire

$$\begin{array}{lllllll} S & \rightarrow & E\,\$ & & E & \rightarrow & id \\ E & \rightarrow & E + E & & & & \end{array}$$

plutôt que disambiguer en

$$\begin{array}{lllllll} S & \rightarrow & E\,\$ & & E & \rightarrow & T \\ E & \rightarrow & T + E & & T & \rightarrow & id \end{array}$$

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, directive %prec

On peut aussi introduire des terminaux fictifs, juste pour les précédences:

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

```
%{
%}
%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)

$$
\begin{array}{lcllcl}
S & \rightarrow & E\,\$ & E & \rightarrow & if\ E\ then\ E\ [\ else\ E\ ] \\
E & \rightarrow & ID & & &
\end{array}
$$

L'ambiguïté produit le conflit:

```
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*, ce qui est bien dans ce cas, 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 {};
%%
```

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

```
11: shift/reduce conflict (shift 7, reduce 2) on
11: shift/reduce conflict (shift 8, reduce 2) on
11: shift/reduce conflict (shift 9, reduce 2) on
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
```

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 {};
%%
```