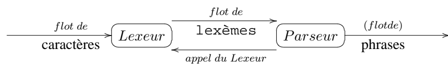
Actions sémantiques: OcamlLex, LL, OcamlYacc

- Bref aperçu des grammaires attribuées
- Grammaires S-attribuées en OcamlYacc et LR
- L'exemple classique: la calculette en OcamlYacc/OcamlLex
- Grammaires L-attribuées en OcamlYacc et LR
- Grammaires L-attribuées dans le cadre LL

Schéma de principe

On a vu construire un analyseur lexical (OcamlLex), des analyseurs LL (à la main) et LR (via OcamlYacc), mais ces constructions se limitent à répondre OUI ou NON à la question "est-ce ce mot partie du langage régulier ou algébrique suivant?"

Nous devons maintenant enrichir cette réponse avec de l'information sémantique (c'est à dire avec des valeurs qui décrivent le mot qui a été reconnu, ou une autre information calculée à partir de ce mot). En général, la première partie de notre compilateur (le "front-end") aura la structure du schéma suivant



où *lexème* et *phrase* peuvent contenir ou simplement être des informations sémantiques choisies par qui réalise le compilateur.

Valeurs sémantiques dans OcamlLex

Dans la définition de l'analyseur pour OcamlLex, on a déjà vu qu'il y a une partie *action* qui est exécutée dés qu'un lexème est reconnu

```
{ prologue }
let ident = regexp ...
rule etatinitial =
  parse regexp { action }
       | ...
and ...
{ epilogue }
```

Si cette action produit une valeur, cette valeur est retournée comme résultat de la fonction d'analyse. Cela implique que toutes les actions doivent avoir le même type.

Un exemple: le lexeur pour une calculette

On retourne des tokens pouvant contenir des valeurs sémantiques, comme le *int* dans *INT of int*:

```
(* File CalcLex.mll *)
{type token = INT of int | LPAREN | RPAREN | PLUS | MINUS | TIMES | DIV | EOL;;
 exception Eof;;
 let print_token = function  . . . }

rule token = parse
[' ' '\t']     { token lexbuf }     (* skip blanks *)
| ['\n' ]        { EOL }
| ['0'-'9']+     { INT(int_of_string(Lexing.lexeme lexbuf)) }
| '+'            { PLUS }
| '-'            { MINUS }
| '*'            { TIMES }
| '/'            { DIV }
| '('            { LPAREN }
| ')'            { RPAREN }
| eof            { raise Eof }
{ let _ = try
    let lexbuf = Lexing.from_channel stdin in
        while true do print_token (token lexbuf); done
  with Eof -> print_newline(); exit 0 }
```

Grammaires attribuées

Il existe un cadre déclaratif général pour décrire des valeurs sémantiques associés à des grammaires libres de contexte: les *grammaires attribuées*, introduites par Donald E. Knuth dans

Semantics of context-free languages,
Mathematical Systems Theory, 2(2), 127– 145, (1968).

Idée: on définit un certain nombre d'*attributs* qui sont attachés aux symboles terminaux et non terminaux, et des *règles* de calcul qui sont attachèes aux productions de la grammaires.

Sur un arbre d'analyse donné, ces règles, appliquées dans un ordre cohèrent, donneront la valeur des attributs sur chaque noeud de l'arbre.

En général, on s'intéresse aux valeurs des attributs de la racine.

Attributs herités et synthetisés

On écrit `attr(A)` ou `A.attr` pour l'attribut *attr* attaché au non terminal A. On distingue les attributs en deux classes:

synthétisé : un attribut *attr* est synthétisé si pour une production $A \rightarrow X_1 \ldots X_n$ l'attribut $attr(A)$ est calculé à partir des attributs des X_i
Il monte *du bas vers le haut* dans l'arbre.
Pour les terminaux, c'est la valeur restituée par OcamlLex.

hérité : un attribut *attr* est hérité si pour une production $A \rightarrow X_1 \ldots X_n$ l'attribut $attr(X_i)$ est calculé à partir des attributs de A et/ou des X_j $(j \neq i)$
Il descend *du haut vers le bas* dans l'arbre.

Définition 1.1 *Une grammaire attribuée est bien formée si chaque attribut est soit synthetisé, soit hérité, mais pas les deux.*

Grammaires attribuées, exemple classique

La grammaire attribuée suivante montre des attributs hérités (typeh) et synthétisés (types, vallex, tab)

$$
\begin{array}{llll}
D & \rightarrow & T\ L & \{typeh(L) := types(T); tab(D) := tab(L)\} \\
T & \rightarrow & int & \{types(T) := int\} \\
T & \rightarrow & float & \{types(T) := float\} \\
L & \rightarrow & L_1, id & \{typeh(L_1) := typeh(L)\} \\
 & & & \{tab(L) := add(vallex(id), typeh(L), tab(L_1))\} \\
L & \rightarrow & id & \{tab(L) := add(vallex(id), typeh(L), emptytable)\}
\end{array}
$$

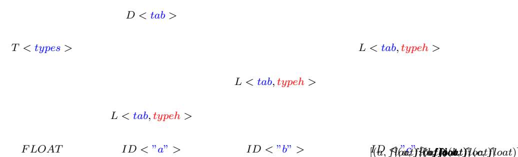
Voyons l'exemple `float a,b,c` que OcamlLex transforme en

FLOAT ID("a") ID("b") ID("c")

avec les bonnes valeurs synthétisées pour les terminaux ID.

Graphe de dependance et calcul des attributs de l'exemple

On mets une flèche entre un attribut a et a' si a sert à calculer a'.
L'ordre de calcul doit respecter les flèches: s'il y a un cycle, pas de résultat[1].



Grammaires S-attribuées et L-attribuées

S'il n'y a pas de cycles dans le graphe de dépendances d'une grammaire attribuée, on peut trouver un ordre de calcul cohérent des attributs, mais cela peut-être difficile et cher en temps de calcul. On s'intéresse alors a deux sous-classes importantes:

S-attribuées une grammaire est S-attribuée si tous les attributs sont synthetisés.

On peut tout calculer bottom-up avec un analyseur LR.

L-attribuées une grammaire est L-attribuée si pour tout symbole X_i de toute production $A \rightarrow X_1 \ldots X_n$, tous les attributs hérités de X_i dependent seulement des attributs des X_j $(j < i)$ et des attributs hérités de A.

N.B.: S-attribuées ⊂ L-attribuées.

On peut tout calculer dans une visite en profondeur d'abord.

[1] À moins de calculer un point fixe,mais cela ne nous interesse pas ici

Grammaires S-attribuées en analyse LR

On peut modifier un analyseur ascendant LR pour qu'il stoque sur la pile non seulement les symboles terminaux et non terminaux, mais aussi les attributs synthétisés associés à ces symboles.

Pour cela, il faut que l'analyseur sache associer une valeur à chaque symbole de pile.
Voyons les deux cas de figure:

Symboles terminaux sur la pile

Un symbole terminal est mis sur la pile exclusivement par un décalage, ce qui est fait après avoir obtenu le symbole terminal de l'analyseur lexical.
Maintenant, l'analyseur lexical retourne une valeur sémantique, et c'est bien celle-là qui sera utilisée.

Symboles non-terminaux sur la pile

Un symbole nonterminal est placé sur la pile exclusivement comme conséquence d'une réduction LR, en passant d'une configuration

$$(\alpha X_1 \ldots X_j \quad , \quad a_i \ldots a_n)$$

à une configuration

$$(\alpha Y \quad , \quad a_i \ldots a_n)$$

par une réduction LR par la règle $Y \rightarrow X_1 \ldots X_j$ de la grammaire[2].

Comme notre grammaire est S-attribuée[3], on peut calculer la valeur des attributs de Y en n'utilisant que les attributs des X_i, qui sont déjà présents sur la pile!

Dans `OcamlYacc`, on écrit la règle de calcul en `Ocaml` dans la partie *action*.

Valeurs sémantiques en analyse LR (fin)

Remarque La possibilité d'utiliser des grammaires recursives à gauche ou même ambiguës permet de donner une forme assez naturelle aux actions, à différence de ce qui se passe en analyse LL (voir après)

Remarque L'analyseur modifié empile 3 choses à la fois sur la pile: le symbole (terminal ou non terminal) X, sa valeur v , et l'état courant s de l'automate.

Exemple d'exécution avec valeurs sémantiques

$$
\begin{array}{lllll lllll}
0 & S & \rightarrow & E\ \$ & \{v(S) := v(E)\} & 2 & E & \rightarrow & T & \{v(E) := v(T)\} \\
1 & E & \rightarrow & E + T & \{v(E) := v(E) + v(T)\} & 3 & T & \rightarrow & int & \{v(T) := v(int)\}
\end{array}
$$

	Action			Trans	
	int	$+$	$\$$	E	T
1	$s6$			$g3$	$g2$
2	$r2$	$r2$	$r2$		
3		$s5$	a		

	Action			Trans	
	int	$+$	$\$$	E	T
4	$r1$	$r1$	$r1$		
5	$s6$				$g4$
6	$r3$	$r3$	$r3$		

[2] Ici on utilise X_i pour indiquerdes symboles terminaux ou non terminaux
[3] tous les attributs sont synthétisés

$$
\begin{array}{llrl}
(_1 & , & int(10)+int(7)\ \$) & shift \\
(_1\underline{(int,10)_6} & , & +int(7)\ \$) & reduce, propagation \\
(_1\underline{(T,10)_2} & , & +int(7)\ \$) & reduce, propagation \\
(_1(E,10)_3 & , & +int(7)\ \$) & shift \\
(_1(E,10)_3(+,_)_5 & , & int(7)\ \$) & shift \\
(_1(E,10)_3(+,_)_5\underline{(int,7)_6} & , & \$) & reduce, propagation \\
(_1\underline{(E,10)_3(+,_)_5(T,7)_4} & , & \$) & reduce, calcul \\
(_1(E,17)_3 & , & \$) & accept
\end{array}
$$

Ici on a écrit $(X,v)_s$ pour le triplet constitué du symbole X, la valeur v et l'état s.

Valeurs sémantiques dans OcamlYacc

En OcamlYacc, le bout de code Ocaml dans la partie "action", peut accéder à la valeur (mémorisée sur la pile) du symbole numéro i de la production courante par la notation $i Voilà ce qui devient, formellement, l'exemple précédant:

```
%token<int> INT
%token EOF PLUS
%start s
%type <int> s
%%
s:     e EOF    { $1 };
e:     e PLUS t { $1 + $3⁴ }
     | t        { $1 };
t:     INT      { $1 };
```

Notez le token *INT* qui transporte une valeur sémantique entière.

Un exemple complet: le lexeur de la calculette

```
(* File lexer.mll *)
{
open Parser      (* The type token is defined in parser.mli *)
 exception Eof
}
rule token = parse
[' ' '\t']     { token lexbuf }     (* skip blanks *)
| ['\n' ]        { EOL }
| ['0'-'9']+     { INT(int_of_string(Lexing.lexeme lexbuf)) }
| '+'            { PLUS }
| '-'            { MINUS }
| '*'            { TIMES }
| '/'            { DIV }
| '('            { LPAREN }
| ')'            { RPAREN }
| eof            { raise Eof }
```

Un exemple complet: le parseur de la calculette

[4] Les tokens sans arguments n'ont pas de valeur! Utiliser $2 ici produit une erreur OcamlYacc


```
%token <int> INT
%token PLUS MINUS TIMES DIV LPAREN RPAREN EOL
%left PLUS MINUS        /* lowest precedence */
%left TIMES DIV         /* medium precedence */
%nonassoc UMINUS        /* highest precedence */
%start main             /* the entry point */
%type <int> main
%%
main:  expr EOL                 { $1 };
expr: INT                       { $1 }
      | LPAREN expr RPAREN      { $2 }
      | expr PLUS expr          { $1 + $3 }
      | expr MINUS expr         { $1 - $3 }
      | expr TIMES expr         { $1 * $3 }
      | expr DIV expr           { $1 / $3 }
      | MINUS expr %prec UMINUS { - $2 };
```

Un exemple complet: la calculette

La boucle principale...

```
 (* File calc.ml *)
let _ =
try
let lexbuf = Lexing.from_channel stdin in
while true do
let result = Parser.main Lexer.token lexbuf in
print_int result; print_newline(); flush stdout
done
with Lexer.Eof ->
exit 0
```

Un exemple complet: la calculette (fin)

Le fichier `Makefile` (attention aux tabulations!!!)

```
CAMLC=ocamlc
CAMLLEX=ocamllex
CAMLYACC=ocamlyacc

all: parser.cmi parser.cmo lexer.cmo calc.cmo
        ocamlc -o calc lexer.cmo parser.cmo calc.cmo
clean:
        rm *.cmo *.cmi calc

# generic rules :
#################
.SUFFIXES: .mll .mly .mli .ml .cmi .cmo
.mll.mli:
        $(CAMLLEX) $<
.mll.ml:
        $(CAMLLEX) $<
```



```
.mly.mli:
        $(CAMLYACC) $<
.mly.ml:
        $(CAMLYACC) $<
.mli.cmi:
        $(CAMLC) -c $(FLAGS) $<
.ml.cmo:
        $(CAMLC) -c $(FLAGS) $<
```

Grammaires L-attribuées en LR

Les attributs hérités posent un problème en analyse ascendante: ils démandent d'évaluer des expressions *au mileu* de la reconnaissance d'une poignée[5], ce qui n'est pas prévu par le modèle LR. Yacc pallie au problème en permettant d'écrire des actions au milieu des règles:

$$A \to X_1 \dots X_i \{\$0 = \$-3\} \dots X_n \qquad \{\dots\}$$

qui est une abréviation pour l'introduction d'un marqueur M:

$$
\begin{array}{lll}
A \to & X_1 \dots M X_i \dots X_n & \{\dots\} \\
M \to & \epsilon & \{\$0 = \$-3\}
\end{array}
$$

La nouveauté essentielle est que les $0, $-1, ... manipulent au fond de la pile les valeurs des symboles non encore réduits.

Mais cela peut introduire des conflits shift/reduce[6]!

Attributs hérités et OcamlYacc

En OcamlYacc, on ne permet pas d'actions au milieu des règles, ... parce-que avec un langage fonctionnel on peut mieux faire, il suffit de retourner, à la place des valeurs synthétisés, des fonctions qui les calculent à partir de paramètres qui sont les valeurs herités!

Voyons en pratique comment cela se passe pour la grammaire des déclarations de type *"à la C"*

```
%{
   let add (id,typ,l) = (id,typ)::l;;
   let empty = [];;
%}
%token INT COMMA FLOAT EOF
%token <string> ID

%start s
%type <(string*string) list> s
```

[5] handle

[6] $A \to id\ DA \to id\ E$vs.$A \to \{act1\} id\ DA \to \{act2\} id\ E$


```
%%
s: d EOF       {$1};
d: t l         {$2 $1}; /* $2 est une fonction! */
t: INT         {"int"}
   | FLOAT     {"float"};
l: l COMMA ID { fun t -> add($3,t,($1 t)) }
   | ID        { fun t -> add($1,t,empty)};
```

Valeurs sémantiques en analyse LL

Un analyseur LL descendant construit l'arbre top-down, donc il peut gérer facilement les attributs *hérités* des grammaires L-attribuées, mais le traitement des valeurs sémantiques *synthétisés* est très peu intuitif, parce que on doit travailler avec des grammaires transformées.

Revoyons l'exemple simple de tout à l'heure, mais en version LL.
La grammaire (annotée avec les actions)

$$
\begin{array}{llll}
S & \to & E\ \$ & \{v(S) := v(E)\} \\
E & \to & E\ PLUS\ T & \{v(E) := v(E) + v(T)\} \\
E & \to & T & \{v(E) := v(T)\} \\
T & \to & int & \{v(T) := v(int)\}
\end{array}
$$

n'est pas LL, parce que ell'est recursive à gauche.

Valeurs sémantiques en analyse LL (suite)

Il faut derecursiviser (en passant, on élimine le symbole T qui est redondant)

$$
\begin{array}{lll}
S & \to & E\ \$ \\
E & \to & int\ E' \\
E' & \to & +\ int\ E' | \epsilon
\end{array}
$$

Mais quid des règles de calcul des attributs?

Transformation des attributs par derecursivisation

Voilà la règle: si on a une recursion gauche

$$
\begin{array}{llll}
A_1 & \to & A_2 Y & \{attr(A_1) := g(attr(A_2), attr(Y))\} \\
A & \to & X & \{attr(A) := f(attr(X))\}
\end{array}
$$

pendant la derecursivisation, on transforme aussi les actions, en introduisant un attribut hérité *attrh* en plus de l'attribut synthétisé pour le nouveau nonterminal A' introduit par la transformation

$$
\begin{array}{llll}
A & \to & X A' & \{attrh(A') := f(attr(X)); attr(A) := attr(A'\} \\
A'_1 & \to & Y A'_2 & \{attrh(A'_2) := g(attrh(A'_1), attr(Y)); \\
 & & & \ attr(A'_1) := attr(A'_2))\} \\
A' & \to & \epsilon & \{attr(A') := attrh(A')\}
\end{array}
$$

N.B.: on peut prouver que cette transformation préserve les grammaires L-attribuées et produit le mêmes valeurs pour les attributs des nonterminaux originaux.

Sur l'exemple

La grammaire récursive à gauche

$$\begin{array}{lcll} S & \rightarrow & E\ \$ & \{v(S) := v(E)\} \\ E & \rightarrow & E\ PLUS\ int & \{v(E) := v(E) + v(int)\} \\ E & \rightarrow & int & \{v(E) := v(int)\} \end{array}$$

devient (on introduit un nouvel attribut hérité *vh*)

$$\begin{array}{lcll} S & \rightarrow & E\ \$ & \{v(S) := v(E)\} \\ E & \rightarrow & int\ E' & \{vh(E') := v(int); v(E) := v(E')\} \\ E'_1 & \rightarrow & +\ int\ E'_2 & \{vh(E'_2) := vh(E'_1) + v(int); v(E'_1) := v(E'_2)\} \\ E' & \rightarrow & \epsilon & \{v(E') := vh(E')\} \end{array}$$

En pratique: sur le code Ocaml

Voici l'analyseur LL que l'on avait construit pour cette grammaire, quand on ne s'intéressait pas aux valeurs sémantiques:

```
exception Parse_Error;;
type token = Int | PLUS | Eof;;
let descente gettoken = . . .
 let rec ntS () = (ntE(); check(Eof))
 and ntE () = match !tok with  Int -> (eat(Int); ntE'())
                            |  _   -> raise Parse_Error
 and ntE' () = match !tok with PLUS -> (eat(PLUS);eat(Int); ntE'())
                            |  _      -> () (* transition vide *)
in ntS();;

(* un lexeur bidon vite fait *)
let gettoken_of l = let data = ref l in
    let f () = let t = List.hd !data in data:= List.tl !data; t in f;;
descente (gettoken_of [Int;PLUS;Int;PLUS;Int;Eof]);;
```

Valeurs sémantiques en analyse LL (suite)

Les attributs hérités deviennent des paramètres des fonctions, les attributs synthétisés sont les résultats des fonctions.

```
type token = Int of int| PLUS | Eof;;
 let rec ntS () = (let e = ntE() in (check(Eof);e))
 and ntE () = match !tok with  Int(n) -> (advance(); ntE'(n))
                            |  _   -> raise Parse_Error
 and ntE'(v) = match !tok with
                 PLUS -> (eat(PLUS);
                          match !tok with
                            Int(n) -> ntE'(v+n)
                            _      -> raise Parse_error)
                 |  _      -> (v) (* transition vide *)
in ntS();;
```


Valeurs sémantiques dans LL (fin)

LL est très mal adapté à l'analyse avec valeurs sémantiques de grammaires qui sont naturellement recursives à gauche: on est obligé de transformer les actions pour les adapter aux grammaires derecursivés, ce qui fait perdre de naturalité à l'analyseur, contrairement à ce qui se passe dans le cas LR.

La méthode de transformation de grammaires attribuées vous guide dans la construction de tels analyseurs.