

## Le front-end: du langage source à l'arbre de syntaxe abstraite

*Quelque rappels:*

- ▶ analyse lexicale avec OcamlLex et analyse syntaxique avec OcamlYacc
- ▶ la notion d'arbre de syntaxe abstrait (AST)
- ▶ construction de l'AST avec les actions sémantiques de OCamlYacc
- ▶ CTigre: aperçu du langage du projet, et du front-end fourni
- ▶ Portée des identificateurs et tables des symboles

## Analyse lexicale avec OcamlLex et analyse syntaxique avec OcamlYacc

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

## Rappels sur les langages rationnels

**expressions rationnelles (regular expressions) :**

$\epsilon, +, \cdot, *$

**automates finis non déterministes :**

$$(Q, I, T, \mathcal{F})$$

**automates finis déterministes :**

$$(Q, I, T, \mathcal{F})$$

avec  $\mathcal{F}$  fonctionnelle et  $I$  singleton

**déterminisation :** construction de l'ensemble puissance

**minimalisation :** construction de Moore

**propriétés :** lemme de l'étoile; fermeture par complémentation, intersection, union; décidabilité du langage vide, fini ou infini

### Exemple OcamlLex

```
{exception Eof;;
  let count = ref 0;;
}
let alphanum = [^ ' ' '\t' '\n']
let mot = alphanum alphanum*
rule token = parse
  mot      {count := !count + 1;}
  | _      {}
  | eof    {raise Eof}
{
let _ = try let lexbuf = Lexing.from_channel stdin in
  while true do token lexbuf; done
  with Eof -> print_int !count; print_newline(); exit 0
}
```

### Extensions pour analyseurs lexicaux

Un analyseur lexical doit séparer un flot de caractères en une série de “tokens” (unités lexicales), chacune définie par une expression régulière.

Cela est légèrement différent du problème de la reconnaissance d'un langage rationnel:

- ▶ on recherche le plus long<sup>1</sup> **prefixe** de l'entrée qui corresponde à une des définitions d'unités lexicales, et on doit pouvoir retourner à la fois ce **prefixe** et identifier quelle définition a été trouvée, donc...
- ▶ on augmente l'automate fini avec une information supplémentaire<sup>2</sup> sur les états finaux

<sup>1</sup>on n'échoue pas s'il reste qqe chose après

<sup>2</sup>la règle qui corresponde à l'état; en cas de ambiguïté, on choisit la première en ordre de définition

- ▶ on maintient deux variables supplémentaires: `dernierEtatFinal` et `positionEntreeAuDernier` qu'il faut maintenir à jour.
- ▶ on retourne `dernierEtatFinal` quand l'automate se trouve bloqué.

### Utilisation de OcamlLex

Un fichier source pour OcamlLex a extension `.mll` et a cette structure:

```
{ prologue }
let ident = regexp ...
rule etatinitial1 =
  parse regexp { action }
  | ...
  | regexp { action }
and etatinitial2 =
  parse ...
and ...
{ epilogue }
```

Les commentaires sont délimités par `( * et * )` comme en OCaml.

Prologue et epilogue sont des sections *optionnelles* qui contiennent du code Ocaml arbitraire<sup>3</sup>

### Utilisation de OcamlLex: expressions rationnelles

Un expression de base est l'une des 6 formes suivantes

- '**char**' le caractère dénoté par `char`.
- `-` un caractère quelconque (*mais pas eof*).
- eof** la fin du flot de caractères.
- "**string**" une chaîne de caractères.
- [ **ens-char** ] filtre tout caractère dans l'ensemble `ens-char`, qui peut être une constante `'c'`, un intervalle `'c1' - 'c2'` ou l'union de deux ensembles (ex: `'a' - 'z' 'A' - 'Z'`)
- [ ^ **ens-char** ] tout caractère qui n'est pas dans `ens-char`

### Utilisation de OcamlLex: expressions rationnelles

Les expressions de base peuvent être composées avec les opérateurs suivants

- exp \*** étoile de Kleene
- exp +** 1 ou plus occurrences de `exp`
- exp ?** 1 ou 0 occurrences de `exp`
- exp1 | exp2** une occurrence de `exp1` ou une de `exp2`
- exp1 exp2** une occurrence de `exp1`, puis une de `exp2`
- ( exp )** la même chose que `exp`
- ident** l'expression abrégée de nom `ident`

#### Précédences:

\* et + précèdent ?, qui précède la concaténation, et enfin |

<sup>3</sup> Typiquement: prologue définit des fonctions nécessaires dans les actions epilogue est souvent utilisé pour réaliser un programme "stand-alone"

## Utilisation de OcamlLex: abréviations

Il est possible de donner un nom à des expressions rationnelle qui apparaissent souvent, en écrivant:

```
let ident = regexp
```

entre le prologue et les règles.

Bien entendu, ces definitions *ne peuvent pas être récursives*.

## Utilisation de OcamlLex: flots et états initiaux

Dans `rule token = parse`, `token` est un identificateur Ocaml valide (qui commence par une minuscule), et définit un état initial de l'automate, que l'on peut invoquer sur un flot de caractères.

```
let lexbuf = Lexing.from_channel stdin in
token lexbuf;
```

Construit un flot à partir de l'entrée standard et appelle l'automate sur le flot avec l'état initial "token".

La possibilité d'avoir plusieurs états initiaux est fort pratique pour "changer de mode", ce qui permet de traiter par exemple de façon simple les commentaires imbriqués.

## Les actions

Dans les actions, on peut mettre du code Ocaml quelconque. Le système nous donne quelques fonctions pour interagir avec l'état du lecteur; si `lexbuf` est le nom de notre tampon, alors

**Lexing.lexeme lexbuf** est la chaîne de caractères reconnue

**Lexing.lexeme\_char lexbuf n** est le  $n$ -ième caractère dans la chaîne reconnue (on commence à 0)

**Lexing.lexeme\_start lexbuf** la position dans le flot d'entrée du début de la chaîne reconnue (on commence à 0)

**Lexing.lexeme\_end lexbuf** la position dans le flot d'entrée de la fin de la chaîne reconnue (on commence à 0)

## Utilisation de OcamlLex

1. écrire un fichier `lexer.mll` avec les définitions OcamlLex
2. appeler OcamlLex: `ocamllex lexer.mll`

3. cela produit le fichier `lexer.ml`

4. compiler `lexer.ml`:

▶ s'il est un programme standalone: `ocamlc -o lexer lexer.ml`  
et ensuite on peut l'exécuter en tapant `./lexer`

▶ s'il est un module: `ocamlc -c lexer.ml`

## Un peu de pratique

---

### Commentaires Imbriqués

---

```
{ exception Eof;;
  let level = ref 0;; }
let ouvrecomm = "/*"
let fermecomm = "*/"

rule token = parse
  ouvrecomm {level:=1; comment lexbuf;}
  | _ {print_string(Lexing.lexeme lexbuf);}
  | eof {raise Eof}
and comment = parse
  fermecomm {level := !level -1;
             if !level=0 then token lexbuf else comment lexbuf;}
  | ouvrecomm {level := !level + 1; comment lexbuf;}
  | _ {comment lexbuf; (* glob comments *)}
  | eof { raise Eof;}

{let _ = try let lexbuf = Lexing.from_channel stdin in
  while true do token lexbuf; done
  with Eof -> exit 0 }
```

### Yacc et OcamlYacc

---

Une fois transformé le fichier source en flot de lexèmes, il nous faut reconnaître des *phrases* (le programme tout entier).

Pour cela on utilise des *analyseurs syntaxiques*, dont les plus diffus sont ceux basés sur l'analyse ascendante et les automates à pile LALR(1).

La théorie de l'analyse ascendante vous a déjà été exposée dans le cours de la dernière année de Licence.

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

```
%{
%}
/* un commentaire OcamlYacc! */
%token EOF ID PLUS
%start s
%type <unit> s
%%
La grammaire
    S → E $
    E → T + E | T
    T → id
est décrite dans le fichier
source OcamlYacc à coté:
    s: e EOF {}
    ;
    e: t PLUS e {}
    | t {}
    ;
    t: ID {}
    ;
%%
```

### Structure de la source OcamlYacc

Comme pour OcamlLex, le 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:

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

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

**`%nonassoc symbol...symbol`**

## Description des règles de la grammaire

Les lignes

```
e:      t PLUS e {}4
      | t {}5
;
```

décrivent les deux productions

$$\begin{array}{l} E \rightarrow E + T \\ E \rightarrow T \end{array}$$

Entre accolades on peut mettre du code Ocaml, qui a accès à aux valeurs  $\$1 \dots \$n$  construit par les actions sémantiques associées aux symboles de la partie droite de la production.

## 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`<sup>6</sup> 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`.

---

<sup>4</sup>ici, des actions *sémantiques*

<sup>5</sup>ici aussi

<sup>6</sup>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.

## 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 précedence; si on veut lire  $-1+3*4*5=-4+3+4$  comme  $((-1)+((3*4)*5))=((-4)+3)+4$ , on peut écrire

```

/* precedence      associativité */
%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.

## Exemple

---

Donc pour la grammaire

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

plutôt que disambiguer<sup>7</sup> 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 {};
```

---

S	→	E \$
<sup>7</sup> E	→	T
E	→	T + E
T	→	id



## Précédences, plus en détail

---

Quand OcamlYacc 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 réduit
    - \* 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 ambiguë des expressions arithmétiques avec la déclaration

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

on résout le conflit

```
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 associatif 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 droite):

```
S   →  E $      E   →  id
E   →  E + E    E   →  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

$$\begin{array}{l}
 S \rightarrow E \$ \quad E \rightarrow \text{if } E \text{ then } E [ \text{else } E ] \\
 E \rightarrow id
 \end{array}$$

```

%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)

---


$$\begin{array}{l}
 S \rightarrow E \$ \quad E \rightarrow \text{if } E \text{ then } E [ \text{else } E ] \\
 E \rightarrow id
 \end{array}$$

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*.  
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

```

### 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 {};

```

## Un point sur le choix de conception

---

Il est possible d'écrire un compilateur en mettant tout dans les actions sémantiques du source (Ocaml)Yacc, avec une grammaire attribuée très complexe. . . , mais cela n'est pas idéal:

- ▶ la forme de la grammaire devient une contrainte<sup>8</sup> pour le backend
- ▶ le code de l'analyse sémantique reste lié à l'analyseur syntaxique (difficile à réutiliser pour un autre langage)
- ▶ l'ensemble est peu modulaire et difficile à maintenir

On cherche une interface propre entre analyse syntaxique et back-end.

## Syntaxe abstraite

---

Une fois reconnue une phrase (ou programme) du langage, il est nécessaire de la transformer en une forme adaptée aux phases successives du compilateur, qui vont explorer à plusieurs reprises cette représentation pour vérifier le typage, construire les tables des symboles, calculer la durée de vie des variables, et bien d'autres attributs.

L'arbre de dérivation syntaxique associé à la grammaire utilisée pour l'analyse n'est pas bien adapté, parce qu'il contient un grand nombre de noeuds internes (les non-terminaux de la grammaire), qui n'ont aucun intérêt lors de la visite de la structure.

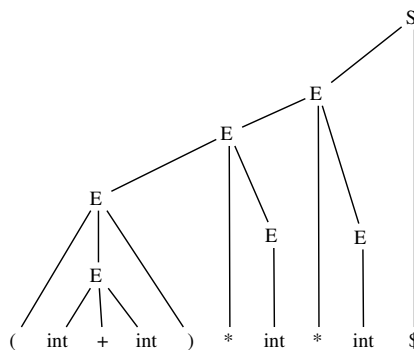
## Syntaxe concrète et arbres d'analyse

---

Exemple: la phrase  $(1 + 2) * 3 * 4$ , avec la grammaire (ambiguë)

$$\begin{array}{lcl} S & \rightarrow & E \$ \\ E & \rightarrow & E + E \\ E & \rightarrow & int \\ E & \rightarrow & E * E \\ E & \rightarrow & (E) \end{array}$$

a comme arbre de *syntaxe concrète*



---

<sup>8</sup>exemple: forward references

## Syntaxe abstraite

---

**Définition:** un *arbre de syntaxe abstraite* est un arbre dont la structure ne garde plus trace des détails de l'analyse, mais seulement de la structure du programme.

Dans un arbre de syntaxe abstraite, construit à partir d'un arbre syntaxique, on n'a pas besoin de garder trace des terminaux qui explicitent le parenthésage, vu qu'on le connaît déjà grâce à l'arbre syntaxique.

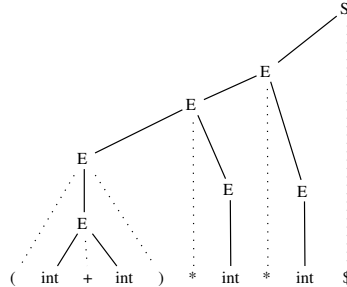
De même, tous les terminaux de la syntaxe concrète (comme les virgules, les points virgules, les guillemets, les mots clefs etc.) qui ont pour but de signaler des constructions particulières, n'ont plus besoin d'apparaître.

## Syntaxe abstraite

---

La définition est un peu floue, mais on peut mieux comprendre en regardant un exemple pour la grammaire de tout à l'heure.

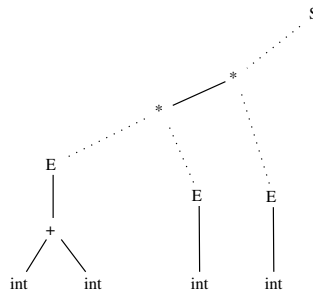
L'arbre syntaxique est très redondant: notamment la seule chose qui nous intéresse dans un noeud  $E(E_1, +, E_2)$  est l'opérateur  $+$  et ses deux fils  $E_1$  et  $E_2$ , donc on peut confondre les noeud  $E$  et  $+$ . De même, on peut oublier les parenthèses et le \$.



## Syntaxe abstraite

---

Ce qui donne

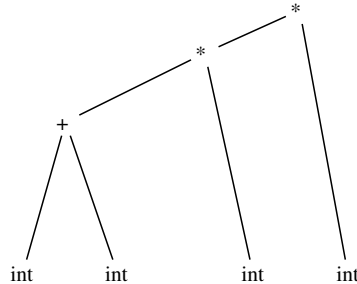


Mais ici, les symboles non terminaux  $E$  et  $S$  n'ont plus aucun intérêt, et on peut les faire disparaître...

## Syntaxe abstraite

---

Pour arriver enfin à



qui est *un* possible arbre de syntaxe abstraite pour l'expression originale...

### Syntaxe abstraite et Ocaml

Les constructeurs de types disponibles dans le langage Ocaml sont très adaptés à la réalisation et manipulation des arbres de syntaxe abstraite.

Voici une possible définition en Ocaml des arbres de syntaxe abstraite obtenus plus haut pour cette grammaire

```
type exp_ast = Int of int
             | Add of exp_ast * exp_ast
             | Mult of exp_ast * exp_ast ;;
```

Le fait que chaque constructeur de type somme est disponible au programmeur permet d'exprimer très simplement l'arbre de syntaxe abstraite pour la phrase  $(1 + 2) * 3 * 4$ :

```
let ast = Mult(Mult(Add(Int(1), Int(2)), Int(3)), Int(4))
```

### Syntaxe abstraite: un exemple complet

Voyons maintenant comment la calculette d'exemple de la dernière fois peut être réalisée en deux phases distinctes, plus modulaires

**construction de l'ast** on retourne comme valeur sémantique un objet `exp_ast`

**évaluation de l'ast** on parcourt l'arbre en procédant à l'évaluation (cela permet par exemple de calculer avec associativité droite des opérateurs définis par commodité comme associatifs à gauche dans la grammaire).

### Syntaxe abstraite: un exemple complet (I)

Le lecteur ne change pas

```
(* 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 }

```

### Syntaxe abstraite: un exemple complet (II)

---

Mais il nous faut maintenant définir un type pour l'arbre de syntaxe abstraite...

```

(* File ast.ml *)
type exp_ast =
  Int of int
  | Add of exp_ast * exp_ast
  | Sub of exp_ast * exp_ast
  | Mult of exp_ast * exp_ast
  | Div of exp_ast * exp_ast
;;

```

### Syntaxe abstraite: un exemple complet (III)

---

Le parseur construit l'arbre de syntaxe abstraite ...

```

%{ open Ast;; %} /* File parser.mly */
%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 <Ast.exp_ast> main
%%
main: expr EOL { $1 };
expr: INT { Int($1) }
  | LPAREN expr RPAREN { $2 }
  | expr PLUS expr { Add($1,$3) }
  | expr MINUS expr { Sub($1,$3) }
  | expr TIMES expr { Mult($1,$3) }
  | expr DIV expr { Div($1,$3) }
  | MINUS expr %prec UMINUS { Sub(Int(0),$2) };

```

### Remarques OcamlYacc

---

*Remarque:* on n'est pas obligé de donner le type de *expr*. En effet, grâce à l'inférence des types de Ocaml, une fois connus les types rendus par les non terminaux de départ, tous les autres types peuvent être *déduits* par le compilateur. C'est un avantage significatif par rapport à ce que l'on doit faire en Yacc ou Bison.

*Remarque:* dans les directives `%token` et `%type`, on doit indiquer le type avec le nom complet (ici `Ast.exp_ast`), et cela même si on a mis la bonne directive `open Ast` dans l'en-tête OcamlYacc.

En effet, l'en tête n'est copié que dans le fichier `parser.ml`, alors que les déclarations produites à partir des directives `%token` et `%type` sont copiés à la fois dans `parser.ml` et dans `parser.mli`.

## Syntaxe abstraite: un exemple complet (IV)

Le fichier principal: on construit l'ast, ensuite on l'évalue

```
(* File calc.ml *)
let rec eval = function
  Int(n) -> n
  | Add(e1,e2) -> (eval e1)+(eval e2)
  | Sub(e1,e2) -> (eval e1)-(eval e2)
  | Mult(e1,e2) -> (eval e1)*(eval e2)
  | Div(e1,e2) -> (eval e1)/(eval e2);;
let _ = try
  let lexbuf = Lexing.from_channel stdin in
  while true do
    let ast = Parser.main Lexer.token lexbuf in
    let result = eval(ast) in print_int result; print_newline()
  done
  with Lexer.Eof -> exit 0;;
```

## Syntaxe abstraite: un exemple complet (fin)

Le fichier Makefile (attention aux tabulations!!!)

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

calc: ast.cmo parser.cmi parser.cmo lexer.cmo calc.cmo
      ocamlc -o calc lexer.cmo ast.cmo parser.cmo calc.cmo
clean:
      rm *.cmo *.cmi calc
# generic rules :
#####
.SUFFIXES: .mll .mly .mli .ml .cmi .cmo .cmx
.mll.mli:
      $(CAMLLEX) $<
.mll.ml:
      $(CAMLLEX) $<
.mly.mli:
      $(CAMLYACC) $<
.mly.ml:
      $(CAMLYACC) $<
.mli.cmi:
      $(CAMLC) -c $(FLAGS) $<
.ml.cmo:
      $(CAMLC) -c $(FLAGS) $<
```



## **Detour: utilisation dans le toplevel Ocaml**

---

Après la compilation, lancez Ocaml, puis tapez

```
#load "ast.cmo";;  
#load "parser.cmo";;  
#load "lexer.cmo";;  
open Ast;;
```

Cela a pour effet de charger dans l'interpreteur les modules compilés ast.cmo, parser.cmo et lexer.cmo (l'ordre est important: dans ce cas, c'est le seul ordre qui ne viole pas les dependences entre modules)

Maintenant on peut écrire:

```
let rec eval = function  
  Int(n) -> n  
    | Add(e1,e2) -> (eval e1)+(eval e2)  
    | Sub(e1,e2) -> (eval e1)-(eval e2)  
    | Mult(e1,e2) -> (eval e1)*(eval e2)  
    | Div(e1,e2) -> (eval e1)/(eval e2);;  
let parse_line () =  
  try  
    let lexbuf = Lexing.from_channel stdin in  
    let ast = Parser.main Lexer.token lexbuf in ast  
  with Lexer.Eof -> failwith "Fin de fichier inattendue";;
```

On peut créer un fichier chargetout.ml

```
(* fichier chargetout.ml *)  
#load "ast.cmo";;  
#load "parser.cmo";;  
#load "lexer.cmo";;  
open Ast;;  
let rec eval = function  
  Int(n) -> n  
    | Add(e1,e2) -> (eval e1)+(eval e2)  
    | Sub(e1,e2) -> (eval e1)-(eval e2)  
    | Mult(e1,e2) -> (eval e1)*(eval e2)  
    | Div(e1,e2) -> (eval e1)/(eval e2);;  
let parse_line () =  
  try let lexbuf = Lexing.from_channel stdin in  
    let ast = Parser.main Lexer.token lexbuf in ast  
  with Lexer.Eof -> failwith "Fin de fichier inattendue";;
```

Ensuite, on lance le toplevel Ocaml et on charge le fichier chargetout.ml avec la directive #use :

```
[dicosmo@localhost]$ ocaml
Objective Caml version 3.07

# #use "chargetout.ml";;
val eval : Ast.exp_ast -> int = <fun>
val parse_line : unit -> Ast.exp_ast = <fun>
```

### Detour: utilisation dans le toplevel Ocaml

Et là, on peut visualiser directement les arbres de syntaxe abstraite:

```
# let a=parse_line();;
1+(2-3)*4/(5--6)
val a : Ast.exp_ast =
  Add
    (Int 1,
     Div (Mult (Sub (Int 2, Int 3), Int 4),
          Sub (Int 5, Sub (Int 0, Int 6))))

# eval a;;
- : int = 1
#
```

### Le front-end que l'on vous donne

On vous fournit dans les fichiers de support du projet le lexeur et le parseur, sous la forme des fichiers

<b>lexer.ml</b>	la sortie de <b>ocamllex lexer.mll</b>
<b>parser.ml</b>	la sortie de <b>ocamlyacc parser.mly</b>

Le parseur contient les actions sémantiques nécessaires pour construire l'arbre de syntaxe abstraite d'un programme CTigre, tel que définit dans `absyntax.ml`

Le fichier `main.ml` vous permet de parser un fichier source CTigre, puis produire et imprimer l'AST associé.

```
exception Erreur_de_syntaxe of int;;
(* compute AST *)
let ast fn =
  let ic = open_in fn in
  let lexbuf = Lexing.from_channel ic in
  try
    let ast = Parser.programme Lexer.token lexbuf in
    (close_in ic); ast
  with Parsing.Parse_error ->
    (close_in ic);
```

```

        raise (Erreur_de_syntaxe(Lexing.lexeme_start lexbuf));;

let main() =
  let past=ref false and . . . in
  let doasource fn = if !past then
    (Printf.printf "\n-----AST-----\n\n";PrintAbsyn.print (stdout,ast
      . . .
    in
  Arg.parse
    [("-ast",Arg.Set past," \tPrint abstract syntax tree");
     . . .
    ]
    (fun fn -> doasource fn) "Usage: ctigre [-ast] file ... file"
in Printexc.catch main ();;

```

### Positions dans le flot d'entrée

Il est important, dans un compilateur réaliste, de pouvoir signaler des erreurs à l'utilisateur avec un certain degré de précision. Pour cela il est important dans les valeurs sémantiques de maintenir:

- ▶ la position initiale et finale dans le flot des caractères ayant donné origine à un terminal
- ▶ la position initiale et finale dans le flot des caractères ayant donné origine à un non terminal

### La syntaxe abstraite de CTigre: I

Pour la réalisation du projet, il nous faut définir une syntaxe abstraite pour le langage CTigre. Comme on souhaite garder trace des positions dans le fichier source, on utilise un module adapté dont voici une esquisse...

```

module Location =
  struct
    type t = int * int
    let pos() = (Parsing.symbol_start(),Parsing.symbol_end())
    let npos n = (Parsing.rhs_start(n),Parsing.rhs_end(n))
    let dummy = (-1,-1) (* par exemple pour le 0 dans *)
                      (* la conversion -i vers 0-i *)
  end

```

## La syntaxe abstraite de CTigre: II

On aura aussi besoin de gérer des tables de symboles, ce qui sera fait dans un module que l'on étoffera et explorera plus avant.

```
module Symbol =
  struct
    type symbol = string
    let symbol n = n
    let name s = s
  end
```

## La syntaxe abstraite de CTigre: 1) types

On trouvera dans la définition de la syntaxe abstraite les différentes composantes du langage:

1) types (notez le traitement des positions!)

```
module Absyn =
  struct
    type symbol = Symbol.symbol

    type_dec = (symbol * core_type)

    and core_type =
      { typ_desc: core_type_desc;
        typ_loc: Location.t }

    and core_type_desc =
      Typ_name of symbol
    | Typ_array of core_type
    | Typ_record of field list
```

## La syntaxe abstraite de CTigre: 2) expressions

```
and exp = { exp_desc: exp_desc; exp_loc: Location.t }

and forexp = { var: symbol; lo: exp; hi: exp;
              dir: direction_flag; for_body: exp }
and direction_flag = Up | Down
and arrayexp = { a_typ: symbol; size: exp; a_init: exp }
and var = SimpleVar of symbol | FieldVar of var * symbol
         | SubscriptVar of var * exp

and exp_desc =
  VarExp of var | NilExp | IntExp of int | StringExp of string
  | Apply of symbol * exp list | RecordExp of (symbol * exp) list
```

```

| SeqExp of exp * exp | IfExp of exp * exp * exp option
| WhileExp of exp * exp | ForExp of forexp
| LetExp of dec list * exp | TypeExp of type_dec list * exp
| ArrayExp of arrayexp | Opexp of oper * exp * exp
| AssignExp of var * exp
and oper = PlusOp | MinusOp | TimesOp | DivideOp
         | EqOp | NeqOp | LtOp | LeOp | GtOp | GeOp

```

### La syntaxe abstraite de CTigre: 3) déclarations

```

and dec =
  FunDec of fundec list
  | VarDec of vardec list
  | TypeDec of type_dec list

and field = {f_name: symbol;
             typ: core_type;
             f_loc: Location.t}

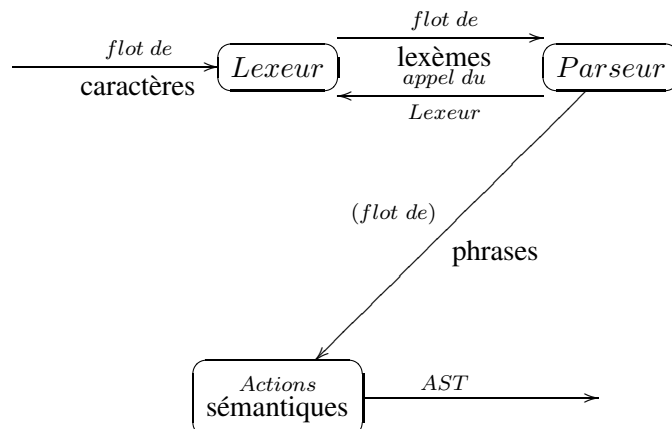
and fundec = {fun_name: symbol;
              params: field list;
              result: core_type option;
              body: exp;
              fun_loc: Location.t}

and vardec = {v_name: symbol;
              var_typ: core_type option;
              init: exp; var_pos: Location.t}

end

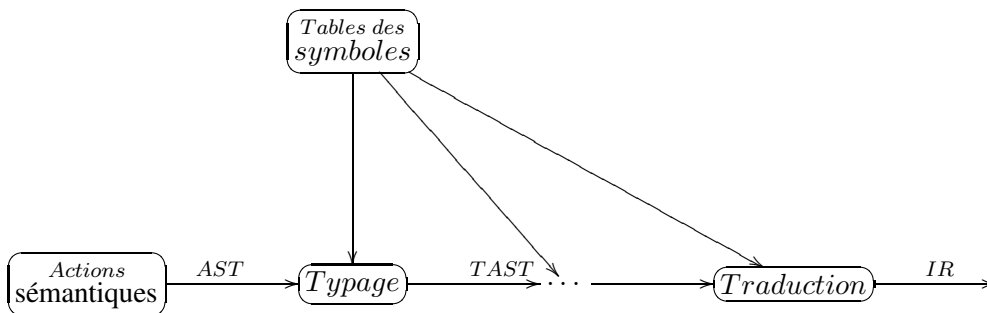
```

### Où on en est



## Où on va

---



## Tables des symboles et portée des identificateurs

---

- ▶ Portée des identificateurs
- ▶ Tables de symboles: structures de données
  - approche impératif: hash et pile de undo
  - approche fonctionnel: arbres binaires équilibrés...
  - en Ocaml

## Portée statique des identificateurs: I

---

Dans les langages modernes, les identificateurs ont presque toujours une portée *statique* limitée au bloc dans lequel ils sont définis.

En CTigre, (comme en Pascal, C++ ou ML, quoique avec une syntaxe différente), considérons un programme contenant le fragment :

```
let a := e1
in e2
```

L'identificateur *a* est lié à la valeur *e1* dans le corps de l'expression *e2*, et cette liaison (*binding* en anglais) est *détruite* dès que l'on sorte de *e2*.

Dans les différentes phases de la compilation, on parcourt l'arbre abstrait pour l'analyse statique (typage, etc.) et la traduction, et on a besoin de savoir, à chaque instant, qu'est la valeur des attributs (type, level, offset, etc.) associés à un identificateur donné.

La table des symboles permet de centraliser cette information.

## Portée des identificateurs: II

---

**Définition 1 (Liaison)** On notera dans la suite  $x \mapsto v$  la liaison entre un identificateur  $x$  et un objet  $v$ .

**Définition 2 (Environnement)** On appelle environnement un ensemble de liaisons. Selon la nature des valeurs associées aux identificateurs, (constantes, cases mémoire, fonctions, types, etc.) on aura divers environnements. Un environnement s'écrira  $\{x_1 \mapsto v_1, \dots, x_n \mapsto v_n\}$ .

Lors de la compilation on a besoin de connaître, quand on rencontre un usage d'un identificateur, quelle est la définition *active* pour cet usage.

## Portée des identificateurs: III

---

Un même nom d'identificateur, défini à l'entrée d'un bloc, peut être *redéfini* dans un bloc plus interne, et donc il ne fera pas forcément référence au même objet. Exemple: considérons le programme

```
0 type a9 = int in
1 let a10 := 3
2 in let f(x11:a) =
3     let b12 :=
4         let a13 := 'A'
5         in ord(a)+x
6     in print("Ord(A)+"); printint(x); print(" = ");
7     printint(b); print(" a vaut "); printint(a)
8 in f(a)
```

l'identificateur `a` défini comme une case mémoire entière contenant l'entier 3 à la ligne 1 est en principe visible dans toute l'expression entre les lignes 3 et 8, mais à la ligne 4 on trouve une redéfinition qui fait en sorte que la liaison entre `a` et 3 est *cachée* dans la ligne 5 par la liaison entre `a` et `'A'` définie à la ligne 4.

Cependant, la redéfinition de la ligne 1 ne cache pas la définition de type de la ligne 0!

## Exemple

---

Suivons l'évolution de l'environnement des liaisons de type et de l'environnement qui donne le type des identificateurs sur le programme de l'exemple

Ligne	Prog	Tenv <sup>14</sup>	TVEnv <sup>15</sup>
0	type a = int in	$\emptyset$	$\emptyset$

<sup>9</sup>nom de type, visible jusqu'à 8

<sup>10</sup>nom de variable, visible en 2,3,6,7,8

<sup>11</sup>nom de variable, visible en 3,4,5,6,7

<sup>12</sup>nom de variable, visible en 4,5,6,7

<sup>13</sup>nom de variable, visible en 5

<sup>14</sup>Valeur du type

<sup>15</sup>Type de l'identificateur

```

1 let a := 3 in      {a ↦ int} ∅
2 let f(x:a) =     {a ↦ int} {a ↦ (l1, int)}
3   let b :=       {a ↦ int} {a ↦ (l1, int); x ↦ (l2, a)}
4     let a := 'A' {a ↦ int} {a ↦ (l1, int); x ↦ (l2, a)}
5     in ord(a)+x  {a ↦ int} {a ↦ (l5, char); x ↦ (l2, a)}
6   in . . .       {a ↦ int} {a ↦ (l1, int); x ↦ (l2, a); b ↦ (l3, int)}
7
8 in f(a)           {a ↦ int} {a ↦ (l1, int)}

```

### Portée des identificateurs: IV

À retenir pour les langages avec structures de blocs:

**portée statique** la portée d'une définition d'identificateur (variable, fonction, procédure, type), i.e. la partie du programme où la liaison pour cet identificateur créée par la définition est active, peut-être obtenu *statiquement*<sup>16</sup>.

**redéfinitions/hiding** une liaison peut être *cachée* temporairement par une nouvelle liaison *de même nature* pour un même identificateur, mais seulement dans un bloc plus interne du bloc de la première définition.

**LIFO** l'ordre dans lequel les liaisons sont introduites est détruites est du genre "Last In First Out", ce qui suggère une pile comme structure de donnée adaptée<sup>17</sup>.

### Portée des identificateurs: structures de données

On doit trouver des structures de données adaptées à la mise en oeuvre des environnements lors de la compilation.

Voilà nos *desiderata*:

- ▶ accès rapide aux liaisons par le nom de l'identificateur
- ▶ gestion facile de l'évolution des environnements, notamment de l'opération d'ajout d'une liaison et de suppression d'une liaison en restaurant l'(éventuelle) liaison précédente

**Définition 3 (Somme d'environnements)** Si  $\sigma$  est un environnement, on écrit  $\sigma + \{x \mapsto v\}$  pour l'environnement qui associe  $v$  à  $x$ , et coïncide avec l'environnement  $\sigma$  sinon.

On écrira  $\sigma_1 + \sigma_2$  pour l'environnement qui contient les liaisons de  $\sigma_1$  et  $\sigma_2$ , mais en donnant priorité à celles de  $\sigma_2$ .

N.B.: l'opération  $+$  ci-dessus n'est pas commutative.

### Solution impérative

On utilise une *table de hachage* spécifique, avec une pile de *undo*.

**entrée dans un bloc** on empile un marqueur, puis on insère les définitions locales

<sup>16</sup> en analysant le texte du programme

<sup>17</sup> tant à la compilation qu'à l'exécution



**insertion** on ajoute la valeur en tête de la liste correspondante à la clef dans la table et on empile la clef sur la pile de undo.

**sortie du bloc** on dépile et on supprime toutes les clefs jusqu'au marqueur de bloc

**suppression** en enlevant l'élément en tête de liste.

Cela peut être mis en place dans un module qui fournit les primitives suivantes:

- ▶ `insert` qui fait à la fois l'insertion en table de hachage et sur la pile de undo,
- ▶ `begin_scope`, qui met le marqueur sur la pile d'undo,
- ▶ `end_scope`, qui dépile les liaisons jusqu'au marqueur compris et les enlève de la table.

C'est l'approche de `gcc`.

### Solution fonctionnelle pure

On utilise des arbres binaires équilibrés, sans pile de *undo*.

**entrée dans un bloc** : on construit un nouvel environnement à partir du précédent,

**sortie d'un bloc** : on abandonne le nouvel environnement et on utilise l'ancien

```
let checktype tenv venv = fonction
  . . .
  | LetExp(VarDecl([x,v]),e) ->
      let newvenv=insert(venv,x,type(v))
      in checktype tenv newvenv e
  | SeqExp(e1,e2)             -> checktype tenv venv e1;
                                checktype tenv venv18 e2
  . . .
```

Pour que cela soit efficace, il faut savoir construire l'objet modifié sans trop "copier".

### Interface du module de la table des symboles

```
module Symbol =
  struct
    exception Not_Found
    type symbol =
    type 'a table =
    let symbol n =
    let name s =
    let mkempty () =
    let add s table =
    let find n table =
    . . .
  end
```

---

<sup>18</sup>réutilisation de l'ancien env

## En Ocaml

---

Le compilateur Ocaml utilise l'approche fonctionnel, (mais avec quelques subtilités).

Module `typing/ident.ml`:

```
type 'a tbl =
  Empty
  | Node of 'a tbl * 'a data * 'a tbl * int

let mknnode l d r =
  let hl = match l with Empty -> 0 | Node(_,_,_,h) -> h
  and hr = match r with Empty -> 0 | Node(_,_,_,h) -> h in
  Node(l, d, r, (if hl >= hr then hl + 1 else hr + 1))

let balance l d r =
  let hl = match l with Empty -> 0 | Node(_,_,_,h) -> h
  and hr = match r with Empty -> 0 | Node(_,_,_,h) -> h in
  if hl > hr + 1 then
    match l with
    | Node (ll, ld, lr, _)
      when (match ll with Empty -> 0 | Node(_,_,_,h) -> h) >=
            (match lr with Empty -> 0 | Node(_,_,_,h) -> h) ->
            mknnode ll ld (mknnode lr d r)
    | Node (ll, ld, Node(lrl, lrd, lrr, _), _) ->
            mknnode (mknnode ll ld lrl) lrd (mknnode lrr d r)
    | _ -> assert false
  else if hr > hl + 1 then
    match r with
    | Node (rl, rd, rr, _)
      when (match rr with Empty -> 0 | Node(_,_,_,h) -> h) >=
            (match rl with Empty -> 0 | Node(_,_,_,h) -> h) ->
            mknnode (mknnode l d rl) rd rr
    | Node (Node (rll, rld, rlr, _), rd, rr, _) ->
            mknnode (mknnode l d rll) rld (mknnode rlr rd rr)
    | _ -> assert false
  else
    mknnode l d r

let rec add id data = function
  Empty ->
    Node(Empty, {ident = id; data = data; previous = None}, Empty, 1)
  | Node(l, k, r, h) ->
    let c = compare id.name k.ident.name in
    if c = 0 then
      Node(l, {ident = id; data = data; previous = Some k}, r, h)
    else if c < 0 then
      balance (add id data l) k r
```

```
        else
            balance l k (add id data r)

let rec find_name name = function
  Empty ->
    raise Not_found
  | Node(l, k, r, _) ->
    let c = compare name k.ident.name in
    if c = 0 then
      k.data
    else
      find_name name (if c < 0 then l else r)
```