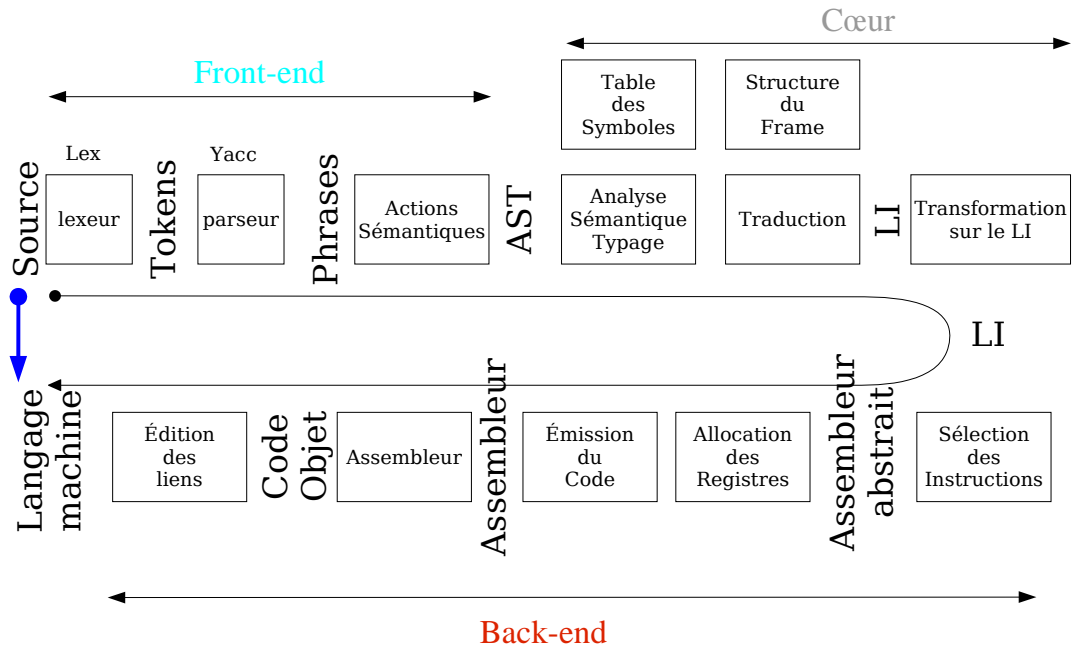


Chapitre 4

Analyse lexicale, analyse syntaxique

Sommaire

4.1	Analyse lexicale	3
4.1.1	Rappels sur les langages rationnels	3
4.1.2	OCamlLex	4
4.2	Analyse syntaxique	7
4.2.1	Théorie	7
4.2.2	OCamlYacc	9
4.3	L'arbre de syntaxe abstraite	15
4.3.1	Représentation du programme	15
4.3.2	Positions	22
4.4	Projet : Le langage CTigre	23
4.4.1	Le front-end que l'on vous fournit	23
4.4.2	CTigre	24
4.4.3	La syntaxe abstraite de CTigre	25



Syntaxe des langages de programmation

Exemple de Fortran (1957 : 18 années-hommes). Le programme suivant affiche les entiers de 1 à 5 :

```

1    PROGRAM test
50   DO 100 I = 1,5
60       WRITE (*,*) I
100  CONTINUE
1000 END

```

Le programme suivant affiche 1.5

```

1    PROGRAM test
50   DO 100 I = 1.5
60   WRITE (*,*) DO 100 I
1000 END

```

Pas de séparation lexical/syntaxique

Aperçu général

Analyse lexicale (Lex/OcamlLex) : Reconnaissance des unités lexicales (= lexèmes ou tokens).
Chaque token est défini par une *expression rationnelle* (= *expression régulière*), reconnue par un *automate fini*

Analyse syntaxique (Bison/Yacc/OcamlYacc/Menhir) : Construction de l'arbre de syntaxe abstraite (AST) à partir de la suite de tokens envoyée par Lex à l'aide d'un automate à pile.
OcamlYacc reconnaît les langages LALR(1) + règles de précedence en cas de conflit
Menhir va jusqu'à LR(1), ce qui évite beaucoup de conflits

OcamlLex et OcamlYacc (ou Menhir) génèrent automatiquement des programmes OCaml qui implémentent les automates.

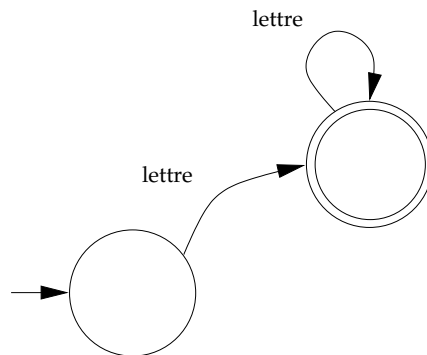
4.1 Analyse lexicale

4.1.1 Rappels sur les langages rationnels

expressions rationnelles (regular expressions)

$\epsilon, +, \cdot, *$

reconnues par les automates finis. Exemple :



automates finis non-déterministes :

$(\Sigma, E, I, T, \mathcal{F})$

avec Σ alphabet, E états, $I \subset E$ états initiaux, $T \subset E$ états terminaux, $\mathcal{F} : E \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(E)$;

automates finis déterministes :

$(\Sigma, E, I, T, \mathcal{F})$

avec $\mathcal{F} : E \times \Sigma \rightarrow E$ et I singleton.

déterminisation : construction de l'ensemble puissance ;

minimisation : construction de Moore ;

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

4.1.2 OCamlLex

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 (on n'échoue pas s'il reste quelque chose après) **préfixe** de l'entrée qui corresponde à une des définitions d'unités lexicales, et l'on doit pouvoir retourner à la fois ce préfixe et identifier quelle définition a été trouvée, donc...
- on augmente l'automate fini avec une information supplémentaire (la règle qui correspond à l'état; en cas d'ambiguïté, on choisit la première dans l'ordre de définition) sur les états finaux
- on maintient deux variables supplémentaires : `dernierEtatFinal` et `positionEntreeAuDernierEtatFinal`, 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 ...  
{ épilogue }
```

Les commentaires sont délimités par (* et *) comme en OCaml. *prologue* et *épilogue* sont des sections *optionnelles* qui contiennent du code Ocaml arbitraire (Typiquement : prologue définit des fonctions nécessaires dans les actions, et épilogue est souvent utilisé pour réaliser un programme indépendant, si l'on ne souhaite pas utiliser OcamlYacc.)

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

Les expressions de base peuvent être composées avec les opérateurs suivants

- exp* étoile de Kleene
 - exp+ 1 ou plusieurs occurrences de exp
 - exp? 1 ou 0 occurrences de exp
 - exp1 exp2 une occurrence de exp1, puis une de exp2
 - exp1 | exp2 une occurrence de exp1 ou une de exp2
 - (exp) la même chose que exp
 - ident l'expression abrégée de nom ident
- Précédence : * et + précèdent ?, qui précède la concaténation, et enfin |

Utilisation de OcamlLex : abréviations

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

```
let ident = regexp
```

entre le prologue et les règles.

Bien entendu, ces définitions *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 lexeur ; 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 }
```

4.2 Analyse syntaxique

4.2.1 Théorie

Une fois transformé le fichier source en flot de lexèmes, il nous faut reconnaître des *phrases* (l'agencement des lexèmes pour former un programme). Pour cela on utilise des *analyseurs syntaxiques*.

La syntaxe des langages que nous voulons compiler est définie par une *grammaire*. Par exemple :

```
PROGRAMME → eof
PROGRAMME → PROGRAMME' eof
PROGRAMME' → DÉFINITION PROGRAMME'
DÉFINITION → let ident = EXPRESSION
EXPRESSION → ...
```

Ici (et dans la suite) les mots en majuscules sont appelés des *non-terminaux*, et les mots en minuscule sont des lexèmes (ou *terminaux*). Chaque ligne est appelée *règle de grammaire* ou *production* de la grammaire.

Une grammaire est dite *ambiguë* lorsqu'il existe une phrase (un programme) qui peut être obtenu de deux façons différentes à partir de l'axiome.

Les automates finis (expressions rationnelles) correspondent aux grammaires *rationnelles*, c'est-à-dire celles dont les parties à droite des flèches sont de la forme a ou bA (terminal ou terminal suivi d'un non-terminal). Malheureusement ceci n'est pas assez puissant pour des langages de programmation. Les langages définis par des grammaires quelconques (avec un non-terminal et un seul à gauche des flèches) sont appelés langages *algébriques* ou *non-contextuels* (*context-free*). Ils sont reconnus par les *automates à pile*. Ces automates utilisent une zone de mémoire organisée en pile, qui permet de sauver des informations.

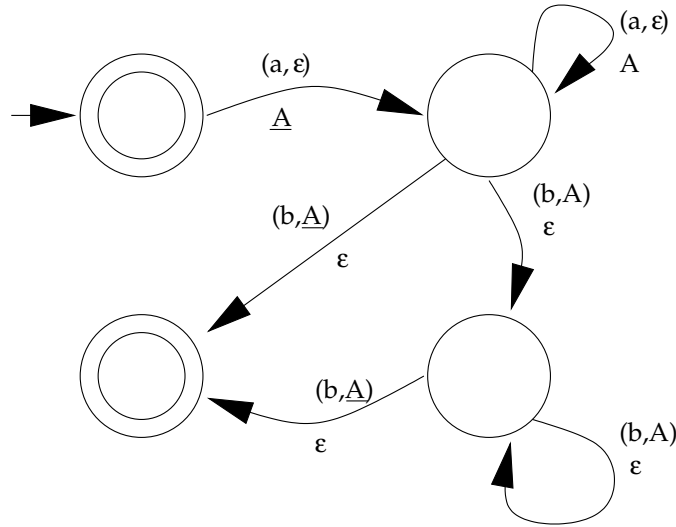
- Le choix d'une transition peut dépendre de la valeur au sommet de la pile (pop)
- et une transition peut entraîner un ou plusieurs empilements (push).

Formellement, un automate à pile peut être défini comme un 7-uplet

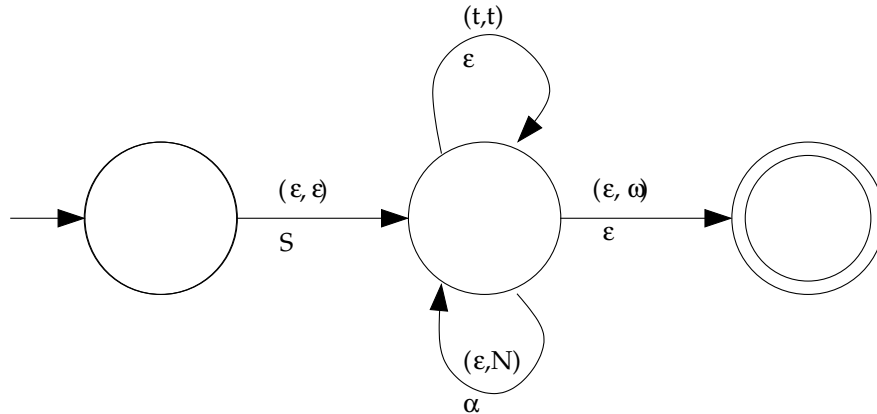
$$(\Sigma, \Phi, \omega, E, I, T, \mathcal{F})$$

avec Σ alphabet d'entrée, Φ alphabet de pile, ω symbole de pile vide, E états, $i \in E$ état initial,

$T \subset E$ états terminaux, $\mathcal{F} : E \times (\Sigma \cup \{\epsilon\}) \times (\Phi \cup \{\epsilon\}) \rightarrow \mathcal{P}(E \times \Phi^*)$ relation de transition.



Notons que ces automates sont non-déterministes. En fait il est assez facile de construire un automate à pile non-déterministe à partir d'une grammaire. Par exemple :



Sur cet automate, l'alphabet de pile est composé des terminaux et non-terminaux. Sur le dessin, N est un non-terminal quelconque et α est un mot tel que $N \rightarrow \alpha$ est une production de la grammaire. S est l'axiome de notre grammaire (le point de départ). On commence par le mettre sur la pile, puis on essaie toutes les règles : dès qu'il y a un non-terminal en tête de pile, on peut le remplacer par la partie droite d'une des règles. S'il y a un terminal sur la pile, alors on le dépile s'il est identique au lexème lu en entrée. S'il existe un chemin qui arrive à une pile vide après avoir lu tout le programme en entrée, alors le programme est valide.

Le gros problème vient du fait que cet automate est très fortement non-déterministe et ne peut être utilisé dans la réalité. On utilise donc des variantes, qui peuvent être classées en deux groupes : *analyse descendante* et *analyse ascendante*. Ces théories vous ont déjà été exposées dans le cours de la dernière année de Licence. Voici juste l'essentiel.

L'analyse descendante part du symbole de départ S et tente d'appliquer les règles pour obtenir l'entrée. On aboutit à la dérivation gauche (c'est à dire le mot reconnu est celui qui aurait

été obtenu en appliquant toujours les règles au non-terminal le plus à gauche). Exemple : $LL(k)$ (lecture de gauche à droite, dérivation gauche, en s'autorisant à lire k lexèmes à l'avance pour faire les choix de transition).

Certains analyseurs implémentent $LL(1)$. Il faut adapter parfois beaucoup la grammaire pour qu'elle devienne $LL(1)$, notamment éliminer les règles récursives à gauche.

L'analyse ascendante donne généralement de meilleurs résultats. Elle part de l'entrée et applique les règles de grammaire en sens inverse pour tenter d'arriver à S . À chaque lexème lu en entrée, on avance dans l'automate (opération appelée *shift*, ou *décale*), en essayant de voir quelles règles ont pu être appliquées pour arriver jusque là. Dès que l'on a trouvé une règle qui convient, on la choisit (opération appelée *reduce* ou *réduit*). On obtient la dérivation droite du mot.

Parfois, on a plusieurs possibilités dans un état donné (par exemple un *reduce* et un *shift* sont possibles). On appelle cela un *conflit*. Il existe plusieurs techniques pour lever les conflits : règles de priorité ou bien lire un lexème à l'avance agir en conséquence.

S'il reste des conflits (ce qui est généralement le cas), on est obligé d'adapter la grammaire pour obtenir quelque chose d'équivalent qui évite le problème.

Exemples d'analyseurs ascendants : $LR(k)$ (lecture de gauche à droite, dérivation droite, k lexèmes lus à l'avance). $LR(0)$ est assez contraignante. $SLR(1)$ et $LALR(1)$ sont des intermédiaires entre $LR(0)$ et $LR(1)$, qui permettent de lever certains conflits de $LR(0)$ mais pas tous. Les analyseurs les plus diffus sont ceux basés sur les automates à pile $LALR(1)$.

La construction des tables $LALR(1)$ pour un vrai langage produit des automates avec plusieurs centaines d'états (la 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 ou Menhir, qui produisent un analyseur écrit en Ocaml. Ces générateurs d'analyseurs sont fait pour travailler en tandem avec Lex (ou Flex) pour C, et OcamlLex, respectivement.

Camlp4 (le préprocesseur d'OCaml) reconnaît les langages $LL(1)$, OcamlYacc reconnaît les langages $LALR(1)$ + règles de précédence en cas de conflit, Menhir reconnaît les langages $LR(1)$.

4.2.2 OCamlYacc

La grammaire des sommes, en OcamlYacc

La grammaire

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

est décrite dans le fichier source OcamlYacc suivant

```
%{
%}
/* un commentaire OcamlYacc */
%token EOF ID PLUS
%start s
%type <unit> s
%%
```

```

s:  e EOF {}
;
e:  t PLUS e {}
    | t {}
;
t:  ID {}
;
%%

```

Attention : OcamlYacc utilise de fait la convention inverse de nos grammaires : les non-terminaux sont en minuscule et les terminaux sont en majuscule...

Structure du 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 point d'entrée (à différence de Yacc !)

Enfin, précédence et associativité permettent de résoudre des conflits (voir plus loin) :

%left *symbol ... symbol*

%right *symbol ... symbol*

%nonassoc *symbol ... symbol*

Description des règles de la grammaire

Les lignes

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

```

décrivent les deux productions

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

Entre accolades on peut mettre du code Ocaml (actions sémantiques), qui a accès aux valeurs \$1 ... \$n construites 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` contenant tous les tokens déclarés avec la directive `%token`. On peut ensuite utiliser ce fichier dans un fichier d'entrée dans les actions OcamlLex, ce qui permet d'écrire la définition des tokens une seule fois à un seul endroit. Attention : le type `token` est donc créé par OcamlYacc, pas par OcamlLex !

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`.

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

```
ocaml yacc -v exmpl.mly
```

produit en plus le fichier

exmpl.output la description des états et transitions de l'automate LALR(1). Ce fichier est utile pour résoudre les conflits.

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 l'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 élevée 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 désambigüer en

$$\begin{aligned} S &\rightarrow E \$ \\ E &\rightarrow T \\ E &\rightarrow T + E \\ T &\rightarrow id \end{aligned}$$

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 {};
```

Précédence, plus en détail

Quand OcamlYacc trouve un conflit shift/reduce dans un état, et dispose d'une série de déclarations de précédence 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 elle est `%left` on réduit
 - si elle est `%right` on décale
 - si elle est `%nonassoc` on met dans la case une action erreur

Par exemple, dans la grammaire ambiguë des expressions arithmétiques avec la déclaration

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

Le fichier `.output` nous montre un conflit sur un état de l'automate étiqueté par :

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

Ici l'automate ne sait pas si dans cet état il doit faire une réduction (reduce) de la règle $E \rightarrow E + E$ ou bien s'il doit continuer à avancer (shift). En d'autre terme, quelque chose de la forme $a + b \times c$ doit-il être parenthésé $(a + b) \times c$ (reduce) ou bien $a + (b \times c)$ (shift)? Ici MULT a une priorité supérieure à PLUS.

Le conflit suivant est résolu par un reduce, parce que PLUS est déclaré associatif à gauche :


```
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

Ce qui suit 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 ça ne fonctionne pas... (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
```

Il faut faire :

```
%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 (exemple : forward references) 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.

4.3 L'arbre de 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 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.

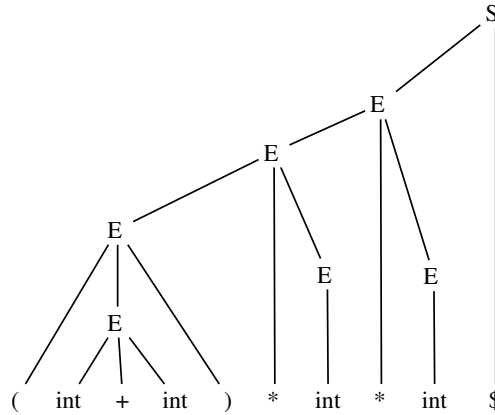
4.3.1 Représentation du programme

Syntaxe concrète et arbres d'analyse

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

$S \rightarrow E \$$	$E \rightarrow E * E$
$E \rightarrow E + E$	$E \rightarrow (E)$
$E \rightarrow int$	

a comme arbre de *syntaxe concrète*



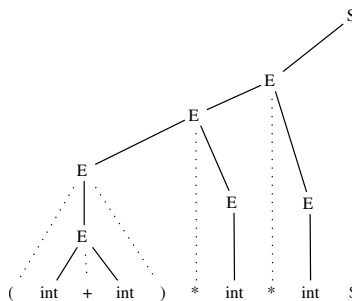
Syntaxe abstraite

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.

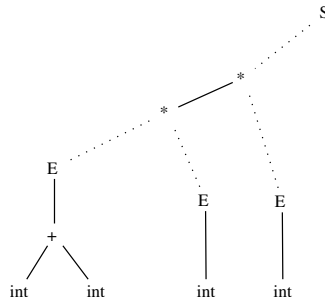
Dans un arbre de syntaxe abstraite, construit à partir d'un arbre syntaxique, on n'a pas besoin de garder trace des terminaux qui explicitent le parenthésage, vu qu'on le connaît déjà grâce à l'arbre syntaxique. De même, tous les terminaux de la syntaxe concrète (comme les virgules, les points virgules, les guillemets, les mots clefs etc.) qui ont pour but de signaler des constructions particulières, n'ont plus besoin d'apparaître.

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 noeuds E et $+$. De même, on peut oublier les parenthèses et le \$.

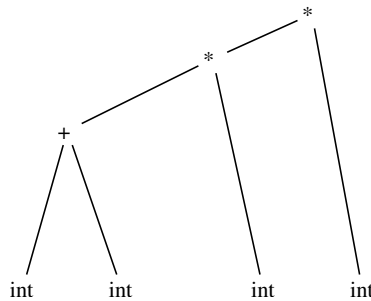


Ce qui donne



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

Pour arriver enfin à



qui est *un* arbre de syntaxe abstraite possible 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 de l'exemple 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).

Le lexeur ne change pas

```
(* File lexer.mli *)
{
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 }
```

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
;;
```

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 sur 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 (suite)

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;;
```

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) $<
```

Détour : 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'interpréteur 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 dépendances 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` :

```
[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>
```

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
#
```

AST en Java

```
abstract class Expr {
    public abstract int eval();}
```

```
class Entier extends Expr {
    public int valeur;
    public Entier(int i) { valeur = i; }
    public int eval() { return valeur; }
}
```

```
class Plus extends Expr {
    public Expr e1, e2;
    public Plus(Expr a, Expr b) { e1 = a; e2 = b; }
    public int eval() { return (e1.eval() + e2.eval()); }
}
```

```
class Mult extends Expr {
    public Expr e1, e2;
    public Mult(Expr a, Expr b) { e1 = a; e2 = b; }
    public int eval() { return (e1.eval() * e2.eval()); }
}
```

```
public abstract class Arithmetique {
    public static void main(String[] arg) {
        Expr e = new Mult(new Plus(new Entier(4), new Entier(5)), new Entier(2))
```

```

        System.out.println(e.eval());
    }
}

```

La distributivité en OCaml :

```

# let rec distribue = function
  | Entier i -> Entier i
  | Plus (a, b) -> Plus (distribue a, distribue b)
  | Mult (a, Plus (b,c)) ->
    Plus (distribue (Mult (a, b)), distribue (Mult (a, c)))
  | Mult (Plus (a, b), c) ->
    Plus (distribue (Mult (a, c)), distribue (Mult (b, c)))
  | Mult (a, b) -> Mult (distribue a, distribue b)
;;
val distribue : expr -> expr = <fun>
# distribue (Mult (Plus (Entier 1, Entier 2), Plus (Entier 3, Entier 4)));;
- : expr =
Plus (Plus (Mult (Entier 1, Entier 3), Mult (Entier 2, Entier 3)),
  Plus (Mult (Entier 1, Entier 4), Mult (Entier 2, Entier 4)))

```

En Java : à vous de l'écrire...

4.3.2 Positions

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

Voici le module utilisé pour gérer les positions dans le projet de 2005 :

```

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

```

Il faut placer les positions partout dans l'AST au moment de sa création. Par exemple :

```

and exp = { exp_desc: exp_desc; exp_loc: Location.t}
and exp_desc =
  VarExp of var | NilExp | IntExp of int | StringExp of string
  | Apply of symbol * exp list | RecordExp of (symbol * exp) list
  | ...

```

Cette année, pour réduire la charge de travail, vous n'aurez pas à gérer les positions (ce qui allège beaucoup le code).

4.4 Projet : Le langage CTigre

4.4.1 Le front-end que l'on vous fournit

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

lexer.ml la sortie de **ocamllex lexer.mll**
parser.ml la sortie de **ocamlyacc parser.mly**

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

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 all_phases = [
  ...
  { key = "-lo";
    doc = " \tPrint attributed abstract syntax tree";
    hdr = "\n#-----ATTRAST---#\n\n";
    flag = ref false;
    action = fun fn -> Printast.print (attrast (ast fn))
  };
  { key = "-int";
    doc = " \tPrint intermediate representation fragments";
    hdr = "\n#-----IR-----#\n\n";
    flag = ref false;
    action = fun fn -> List.iter Frame.print_frag (ir (attrast (ast fn)))
  };
  ...
  { key = "-asm";
    doc = " \tPrint assembler with register allocation";
```

```

    hdr = "\n#-----ASSEM-----#\n\n";
    flag = ref false;
    action = fun fn -> print_asm (instrs simplealloc (stms (ir (attrast (ast fn)
  ]])
}]

type phase = {
  key : string; (* command-line option that trigger this phase *)
  doc : string; (* message for Usage : ... *)
  hdr : string; (* header message displayed before the output of this phase *)
  flag : bool ref; (* should we do this phase ? *)
  action : string -> unit; (* argument is a filename *)
}

let treat_sourcefile fn =
  List.iter
    (fun pha -> if !(pha.flag) then (print_string pha.hdr; pha.action fn))
    all_phases

let main () =
  Arg.parse (get_arglist all_phases) treat_sourcefile
    ("Usage: ctigre "^(get_keys all_phases)^" sourcefile")

```

4.4.2 CTigre

Le langage CTigre est décrit en détail dans l'énoncé du projet CTigre est un langage impératif qui a quelque traits semblables à C (comme le fait que tout expression du langage, même une affectation, a une valeur), et quelques traits semblables à Pascal (déclarations de fonctions locales, typage fort des expressions).

Dans ce langage, il est possible pour le programmeur de :

- définir des *fonctions locales*, avec portée statique des identificateurs comme dans Pascal ou Ocaml
- définir des fonctions *mutuellement récursives*
- définir des *types utilisateur*, à partir des types de base "string", "char" et "int" à l'aide d'*enregistrements* et *vecteurs*
- définir des *types locaux* (avec portée statique)
- définir des *types mutuellement récursifs*

Il a une syntaxe proche de Caml. Voici un exemple :

```

function f(i:int):int =
  var x:=i-1 in
  function g(j:int):int =
    var v:int := x+2 in
    var w:int := v*j in
    w
  in g(i-3)
in f(4)

```

4.4.3 La syntaxe abstraite de CTigre

Pour la réalisation du projet, il nous faut définir une syntaxe abstraite pour le langage CTigre.

On aura besoin de gérer des tables de symboles, ce qui sera fait dans un module `Symbol` que l'on explorera plus tard. Voici un extrait de ce module :

```
type symbol = string
let symbol n = n
let name s = s
```

On trouvera dans la définition de la syntaxe abstraite (`ast.mli`) les différentes composantes du langage :

Types

```
type core_type =
  | Typ_alias of typename
  | Typ_array of typename
  | Typ_record of field list
and field = {fi_name: fieldname; fi_typ: typename}
type typedec = symbol * core_type
```

Expressions

```
type ('v,'f) exp =
  | VarExp of ('v,'f) var
  | NilExp
  | IntExp of int
  | StringExp of string
  | CharExp of char
  | Apply of 'f * ('v,'f) exp list
  | RecordExp of (fieldname * ('v,'f) exp) list * typename
  | SeqExp of ('v,'f) exp * ('v,'f) exp
  | IfExp of ('v,'f) exp * ('v,'f) exp * ('v,'f) exp option
  | WhileExp of ('v,'f) exp * ('v,'f) exp
  | ForExp of ('v,'f) forexp
  | LetVarExp of ('v,'f) vardec list * ('v,'f) exp
  | LetFunExp of ('v,'f) fundec list * ('v,'f) exp
  | TypeExp of typedec list * ('v,'f) exp
  | ArrayExp of ('v,'f) arrayexp
  | Opexp of oper * ('v,'f) exp * ('v,'f) exp
  | AssignExp of ('v,'f) var * ('v,'f) exp

and ('v,'f) forexp = { for_var: 'v; for_lo: ('v,'f) exp; for_hi: ('v,'f) exp;
                      for_dir: direction_flag; for_body: ('v,'f) exp }

and ('v,'f) arrayexp = { a_typ: typename; a_size: ('v,'f) exp;
                        a_init: ('v,'f) exp }

and ('v,'f) vardec = { var_id: 'v; var_typ: typename option;
                      var_init: ('v,'f) exp }
```

```
and ('v,'f) fundec = { fun_id: 'f; fun_params: param list;
                      fun_res: typename option;
                      fun_body: ('v,'f) exp}
```

```
and ('v,'f) var =
  | SimpleVar of 'v
  | FieldVar of ('v,'f) var * fieldname
  | SubscriptVar of ('v,'f) var * ('v,'f) exp
```

```
type direction_flag = Up | Down
```

```
type oper =
  | PlusOp | MinusOp | TimesOp | DivideOp
  | AndOp | OrOp
  | EqOp | NeqOp | LtOp | LeOp | GtOp | GeOp
```

Déclarations

```
and vardec = {var_name: symbol; var_typ: typename option;
              var_pos: varpos; init: exp}
and fundec = {fun_name: symbol; params: param list; result: typename option;
              fun_level: int option; body: exp; num_vars : int option}
and param = {p_name: symbol; p_typ: typename option}
```

Les deux types AST (voir chapitre d'après)

```
type rawexp = (symbol, symbol) exp (* sortie du parseur *)
```

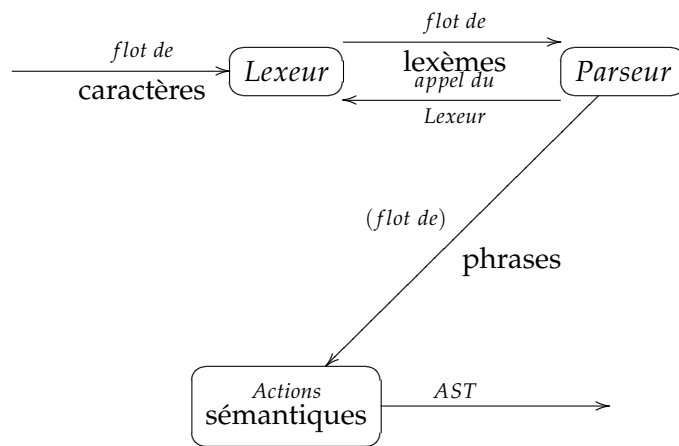
```
type varpos =
  | Stack of int * int (** Allocation in the stack (level, offset) *)
  | Temp of Temp.temp (** Allocation in a temp *)
```

```
type attrvar = { v_name: symbol; mutable v_pos: varpos }
  (** For doing escape analysis, it is convenient to have a mutable [v_pos].
      Otherwise, just forget about it... *)
```

```
type attrfun = { f_name: symbol; f_level: int }
```

```
type attrexp = (attrvar, attrfun) exp (* après calcul de level offset *)
```

Où en est-on ?



Où va-t-on ?

