Grammaires et Analyse Syntaxique - Cours 8 Utiliser Menhir

Ralf Treinen

INFORMATIQUE
Sciences
Université Paris Cité

INSTITUT
DE RECHERCHE
EN INFORMATIQUE
FONDAMENTALE

treinen@irif.fr

25 mars 2022

Générateur d'analyse grammaticale

- Nous avons déjà utilisé un générateur : c'était pour l'analyse lexicale.
- Le principe est le même pour le générateur d'analyse grammaticale, mais il faut comprendre
  1. Comment l'analyse grammaticale va coopérer avec l'analyse lexicale.
  2. Quel est le résultat de l'analyse - est-ce vraiment l'arbre de dérivation qu'on veut ?

Syntaxe concrète

- *Syntaxe concrète* : c'est la définition de la forme correcte de l'écriture (représentation textuelle). Elle est normalement définie par des expressions régulières, et une grammaire hors-contexte.
- La vérification qu'un texte d'entrée suit la syntaxe concrète d'un langage est réalisée en coopération par l'analyse lexicale et l'analyse grammaticale.
- Or, normalement on ne veut pas seulement savoir si l'entrée est correcte ou pas, mais aussi faire quelque chose avec, par exemple :
  - Document XML représentant des données géographiques : afficher une carte.
  - Programme écrit dans un langage de programmation : l'exécuter (cas d'un interpréteur), ou engendrer du code exécutable (cas d'un compilateur).

- On va donc construire une représentation de la structure que les analyses lexicales et grammaticales ont découverte : c'est la *syntaxe abstraite*.
- Le passage de la syntaxe concrète à la syntaxe abstraite a deux fonctions :
  - Vérifier que l'entrée correspond aux règles de la syntaxe concrète,
  - si c'est le cas, construire une représentation en forme de la syntaxe abstraite.
- Il y a ici souvent une *abstraction* : certains détails de la représentation textuelle (en syntaxe concrète) sont omis dans la syntaxe abstraite car ils ne sont pas pertinents pour la suite.
- On essaye d'abstraire de tout qui n'est pas nécessaire pour la suite : simplifier tant que possible !

Abstraction

- Quels sont les détails de la syntaxe concrète dont on peut faire abstraction : ça dépend de ce qu'on compte faire avec !
- Nous avons déjà vu certaines abstractions faites par l'analyse lexicale, dans le contexte des langages de programmation :
  - les espaces (mais attention aux cas où l'incrémentation des lignes indique la structure du programme)
  - les commentaires (mais attention aux cas où les commentaires sont importantes pour la suite, par exemple si on écrit un *pretty printer* de code source)
- Dans un premier temps on peut utiliser l'arbre de dérivation construit par l'analyse grammaticale comme syntaxe abstraite.

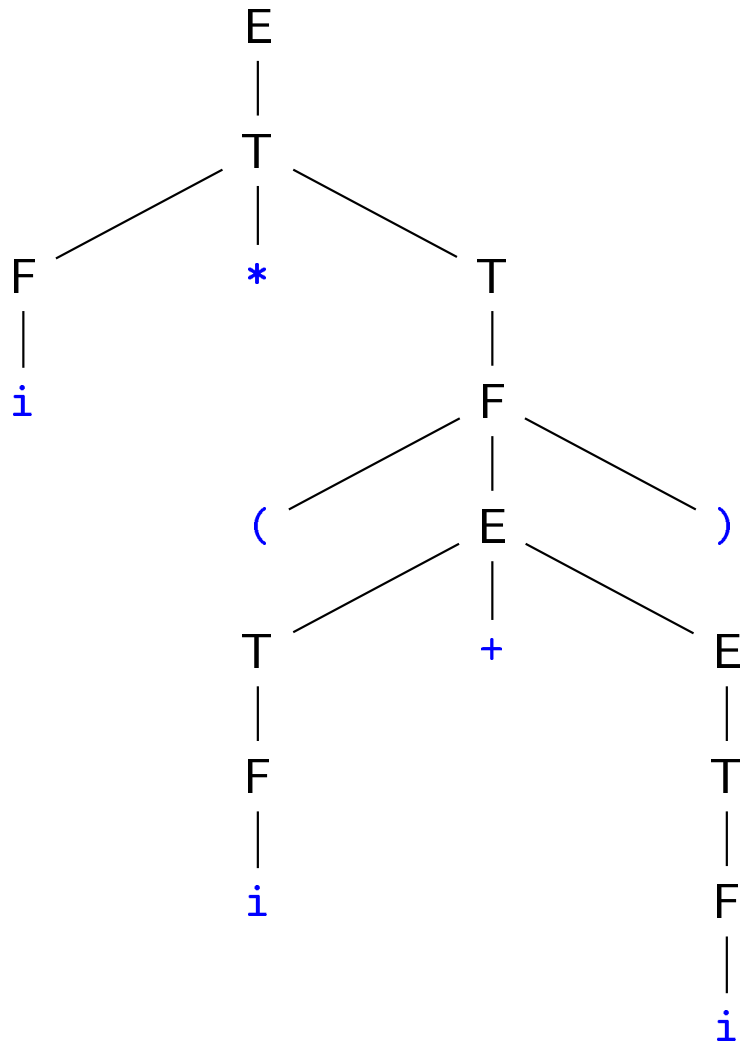
Arbre de dérivation vs. Syntaxe abstraite

- En général, l'arbre de dérivation contient trop d'information.
- Raison : l'arbre de dérivation dit *comment* la structure a été découverte dans le texte d'entrée. Or, c'est seulement la structure elle-même qui est pertinente pour la suite.
- Exemple : grammaire

$$\begin{array}{lcl} \mathsf{E} & \rightarrow & \mathsf{E}\texttt{+}\mathsf{T} \mid \mathsf{T} \\ \mathsf{T} & \rightarrow & \mathsf{T}\texttt{*}\mathsf{F} \mid \mathsf{F} \\ \mathsf{F} & \rightarrow & \texttt{(}\mathsf{E}\texttt{)} \mid \texttt{i} \end{array}$$

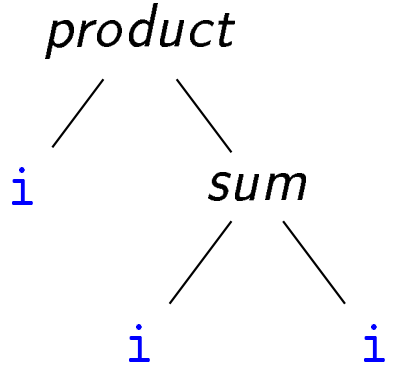
- Arbre de dérivation pour $i * (i + i)$: voir le transparent suivant.

Arbre de dérivation pour $i * (i + i)$

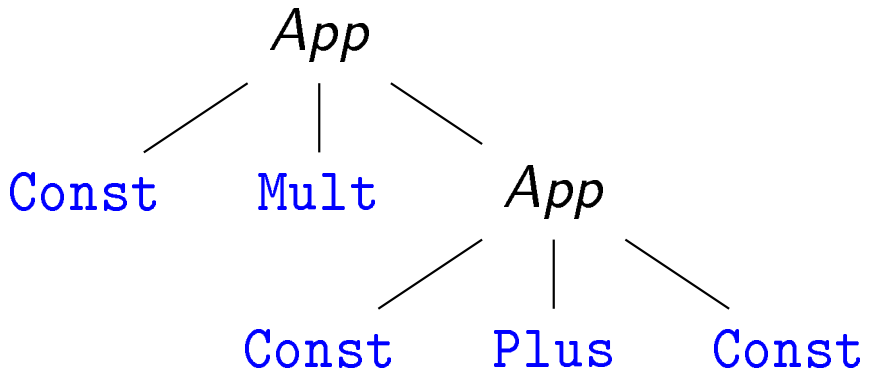


Quel information retenir dans la syntaxe abstraite ?

- La distinction entre E, T et F n'est plus pertinente.
- Les parenthèses ne sont plus pertinentes non plus : elles servent à indiquer la structure dans la syntaxe concrète, mais une fois la structure découverte elle ne servent plus à rien.
- Structure à retenir :



Autre possibilité pour la syntaxe abstraite



- Ici on aura un seule constructeur avec l'opérateur comme argument supplémentaire.
- Solution à préférer si dans la suite tous les opérateurs sont traités de la même façon.

Définition de la Syntaxe Abstraite sur l'exemple

```
(* Syntaxe abstraite des expressions arithmétiques *)

type op = Plus | Mult
type t = Const of int | App of t * op * t
```

Petit historique des générateur d'analyse grammaticale

- 1965 TMG (*TransMogriFier*), du type LL(1)
- Années 70 : YACC (*Yet Another Compiler-Compiler*), utilise LALR(1), qui est entre LR(0) et LR(1). Écrit en C.
- 1985 : Bison, la version GNU de YACC, écrit en C. Produit du code en C, C++, ou Java.
- 1996 ocamlyacc, implémentation de LALR(1). Distribué avec OCaml.
- 2006 : Menhir, implémentation de LR(1). Écrit en OCaml, produit du OCaml.

Exemple : la grammaire des expressions arithmétiques

```
%token <int> INT
%token PLUS TIMES LPARA RPARA EOL
%start <Syntax.t> s
%{ open Syntax %}
%%

s: e1=e EOL              {e1}
e: e1=e PLUS t1=t        {App(e1,Plus,t1)}
 | t1=t                  {t1}
t: t1=t TIMES f1=f       {App(t1,Mult,f1)}
 | f1=f                  {f1}
f: LPARA e1=e RPARA      {e1}
 | i1=INT                {Const(i1)}
```

Structure d'un fichier `.mly`

- Deux (éventuellement trois) parties séparées par %% :
  1. déclarations
  2. règles
  3. (optionnel) : du code OCaml
- Pour l'instant on présente l'utilisation de base seulement.
- Voir la documentation complète de menhir pour plus d'information.

La partie des déclarations (1)

- Déclaration des jetons. Ça correspond à la définition de l'alphabet des terminaux :

  %token t1 t2 t3 ...

  Quand un jeton porte un argument il faut le déclarer avec le type de son argument, par exemple

  %token <**int**> INT

- Les noms des jetons doivent commencer par une majuscule car ils sont traduits, dans le code OCaml engendré, vers les constructeur d'un type OCaml.
- Il est tradition d'écrire les jetons en tout majuscule mais c'est seulement une tradition, et pas une obligation.

La partie des déclarations (2)

- Déclaration de l'axiome, avec le type de la valeur construite, avec la commande %start.
- On peut déclarer plusieurs "axiomes" (qu'il faut plutôt appeler des points d'entrée dans la grammaire).
- Les autres non-terminaux ne sont pas à déclarer, menhir les trouvera sur les côtés gauches des règles.
- Les noms des non-terminaux doivent commencer par une minuscule car ils sont traduits, dans le code OCaml engendré, vers des fonctions OCaml.
- Déclaration d'une entête entre %{ et %}, cette entête sera copiée au début du fichier `.ml` engendré (mais pas dans son fichier interface !).

La partie des règles

- Séquence des règles de la grammaire, groupés par non-terminaux :

```
nonterminal:
| cote-droit-1 {action-1}
....
| cote-droit-n {action-n}
```

  où chaque cote-droit est une séquence de non-terminaux et jetons.
- Les actions sont des expressions OCaml pour calculer la valeur à envoyer (par exemple un morceaux de syntaxe abstraite)
- Sur les côtés droits on peut donner des noms aux valeurs envoyées par des non-terminaux et les jetons ; ces noms peuvent être utilisés dans l'action.

Sur l'exemple

```
e: e1=e PLUS t1=t    {App(e1, Plus, t1)}
```

- e, t : non-terminaux, en minuscules car des identificateurs OCaml
- PLUS : terminal (jeton), en majuscules car constructeur OCaml
- : remplace le symbole $\rightarrow$ des grammaires
- $e1$, $t1$: identificateurs OCAML pour les valeurs sémantiques (des jetons, ou des non-terminaux)
- entre accolades : expression OCaml, peut se servir de $e1$ et $t1$

Styles de syntaxe du fichier menhir

- Ce style de syntaxe des règles est appelé dans le manuel *old syntax* - c'est pourtant le style que nous recommandons, car plus proche des autres générateurs.
- Dans le manuel vous trouvez aussi une *new syntax* expérimentale que nous déconseillons d'utiliser pour ce cours.
- Dans des générateurs historiques (Yacc, Bison) vous trouvez un autre système pour accéder aux valeurs sémantiques par leur position dans la règle ($1, $2, etc.)

Utilisation de base de `menhir`

- menhir parser.mly génère les fichiers parser.ml et parser.mli.
- menhir ––automaton–graph parser.mly produit un graphe de l'automate LR(1) dans le fichier parser.dot
- menhir ––dump parser.mly produit une représentation textuelle de l'automate LR(1) dans le fichier parser.automaton
- menhir ––reference–graph parser.mly produit un graphe de dépendances entre non-terminaux dans le fichier parser.dot
- menhir ––log–grammar 2 parser.mly produit des informations sur la grammaire (annulables, First, Follow, ...)

Le fichier `lexer.mll`

```
{
  open Parser
  exception Error of string
}

rule token = parse
      | [' ' '\t']     { token lexbuf }
      | '\n'           { EOL }
      | ['0'-'9']+ as i      { INT (int_of_string i) }
      | '+'      { PLUS }
      | '*'      { TIMES }
      | '('      { LPARA }
      | ')'      { RPARA }
      | _        { raise (Error (Lexing.lexeme lexbuf)) }
```

Le fichier calc.ml

```
(* programme principal *)

open Syntax

let rec evaluate = function
  | Const i -> i
  | App(e1,Plus,e2) -> (evaluate e1) + (evaluate e2)
  | App(e1,Mult,e2) -> (evaluate e1) * (evaluate e2)
;;

let ast = Parser.s Lexer.token
            (Lexing.from_channel stdin)
    in Printf.printf "Résultat : %i\n" (evaluate ast)
```

Interaction entre ocamllex et menhir

- Les jetons sont déclarés dans le fichier `.mly`
- `menhir` génère un fichier OCaml (et son interface) qui contient en particulier
  - le type `token`
  - une fonction d'analyse syntaxique, qui dans le cas de l'exemple a le type

```
val s : (Lexing.lexbuf -> token)
          -> Lexing.lexbuf -> (Syntax.t)
```

- Donc, au niveau OCaml, le module d'analyse lexicale et le module qui fait appel à l'analyse syntaxique dépendent du module d'analyse grammaticale.

L'automate construit par menhir

- Menhir ajoute une gestion de la fin de l'entrée, ce qui est indiqué dans l'automate par l'ajout d'un nouvel axiome, et d'un symbole # pour la fin de l'entrée.
- Quand c'est possible menhir construit un automate plus simple (SLR(1) ?), mais on peut forcer la construction de l'automate LR(1) selon la méthode présentée dans ce cours, on utilisant l'option ——canonical.
- La différence entre LR(1) et SLR(1) est que l'automate SLR(1) contient les mêmes états que l'automate LR(0), où tous les items [N $\rightarrow \alpha$] sont remplacés par [N $\rightarrow \alpha, \mathrm{Follow}_1$(N)]. L'automate est donc plus petit, mais on perd en précision.

Exemple d'un conflit

- Considérons la grammaire suivante, avec axiome s :

  $s \rightarrow$ expr `EOL`

  expr $\rightarrow$ `IF` expr `THEN` expr | `IF` expr `THEN` expr `ELSE` expr | `ID`

- Dans cet exemple notation menhir : terminaux en majuscules, non-terminaux en minuscules.
- Il s'agit d'un problème classique dans l'analyse syntaxique des langages de programmation : le *dangling else*.
- Le problème est : à quel IF appartient le ELSE :

  `IF ID THEN IF ID THEN ID ELSE ID`

- Regardons ce que menhir en pense.

Fichier menhir pour l'exemple

```
%token IF THEN ELSE ID EOL
%start <unit> s
%%

s: expr EOL                                  {()}
expr: | ID                                   {()}
   | IF expr THEN expr                       {()}
   | IF expr THEN expr ELSE expr             {()}
```

Résultat de menhir sur l'exemple

```
Warning: one state has shift/reduce conflicts.
Warning: one shift/reduce conflict was arbitrarily resolved.
```

- Quand vous voyez un tel message : attention !
- Il faut comprendre d'où vient ce conflit, et le résoudre en modifiant la grammaire, ou ajouter une spécification supplémentaire (voir plus tard) pour contrôler la façon comment le conflit est arbitré.
- Il y a des cas où ce n'est pas possible, par exemple des langages algébriques pour lesquels seulement des grammaires ambiguës existent.
- Comment trouver le conflit ? Menhir peut vous aider beaucoup si vous le lui demandez, en utilisant les bonnes options de menhir.

Dans le fichier parser.automaton

On obtient ce fichier par `menhir --dump parser.mly`.
Extrait du fichier `parser.automaton` :

```
State 5:
## Known stack suffix:
## IF expr THEN expr
## LR(1) items:
expr -> IF expr THEN expr . [ THEN EOL ELSE ]
expr -> IF expr THEN expr . ELSE expr [ THEN EOL ELSE ]
## Transitions:
-- On ELSE shift to state 6
## Reductions:
-- On THEN EOL ELSE
--   reduce production expr -> IF expr THEN expr
** Conflict on ELSE
```

Comprendre le fichier `parser.automaton`

- Attention, quand menhir indique un état il ne met pas tous les items.
- Il y a l'état conflictuel 5 qui contient
  1. [expr $\rightarrow$ `IF` expr `THEN` expr ., {THEN, EOL, ELSE}]
  2. [expr $\rightarrow$ `IF` expr `THEN` expr . `ELSE` expr, {THEN, EOL, ELSE}]
- Il s'agit clairement d'un shift-reduce conflit.
- On peut aussi demander des explications au niveau de la grammaire : `menhir --explain parser.mly`, produit un fichier `parser.conflicts`.

Explication du conflit

```
** Conflict (shift/reduce) in state 5.
** Token involved: ELSE
** This state is reached from s after reading:

IF expr THEN IF expr THEN expr

** The derivations that appear below have the following common factor:
** (The question mark symbol (?) represents the spot where the derivations begin

s
expr EOL
(?)

** In state 5, looking ahead at ELSE, reducing production
** expr -> IF expr THEN expr
** is permitted because of the following sub-derivation:

IF expr THEN expr ELSE expr // lookahead token appears
             IF expr THEN expr .

** In state 5, looking ahead at ELSE, shifting is permitted
** because of the following sub-derivation:

IF expr THEN expr
             IF expr THEN expr . ELSE expr
```

Attention à la résolution arbitraire de conflits

- Quand menhir vous affiche un message comme

```
Warning: 2 states have shift/reduce conflicts.
Warning: 4 shift/reduce conflicts were arbitrarily resolved.
```

- Attention : menhir a arbitrairement résolu les conflits. Le résultat n'est souvent pas ce que vous attendez.
- Exemple : les expressions arithmétiques avec + et *, mais sans gestion propre des priorités.
- En fait cette grammaire est même ambiguë.

Exemple d'un fichier .mly avec conflits

```
%token <int> INT
%token TIMES PLUS LPARA RPARA EOL
%start <Syntax.t> s
%{ open Syntax %}
%%

/* ATTENTION SHIFT-REDUCE CONFLICTS */

s: e1=e EOL                {e1}
e:
   | e1=e TIMES e2=e       {App(e1,Mult,e2)}
   | e1=e PLUS e2=e        {App(e1,Plus,e2)}
   | LPARA e1=e RPARA      {e1}
   | i1=INT                {Const(i1)}
```

Que peut on faire dans ce cas ?

- Première solution : on peut modifier la grammaire, pour rendre la grammaire non ambiguë et gérer les priorités.
- Cela rend la grammaire plus lourde à écrire et à lire (pour les lecteurs humains).
- Deuxième solution : on écrit une grammaire ambiguë avec des informations supplémentaires qui permettent à menhir à résoudre les conflits (et par conséquent à désambiguïser la grammaire).
- Peut seulement arbitrer des conflits shift/reduce.
- Il y a deux types d'informations : associativité et priorité.

Associativité à gauche et à droite

- Un opérateur $\square$ est dit *associatif à gauche* quand une expression

$$x_1 \square x_2 \square x_3$$

et à lire comme

$$(x_1 \square x_2) \square x_3$$

- Un opérateur $\square$ est dit *associatif à droite* quand une expression

$$x_1 \square x_2 \square x_3$$

et à lire comme

$$x_1 \square (x_2 \square x_3)$$

Spécifier associativité en menhir

```
%token <int> INT
%token TIMES PLUS LPARA RPARA EOL
%start <Syntax.t> s
%left PLUS
%right TIMES
%{ open Syntax %}
%%

s: e1=e EOL                 {e1}
e:   e1=e TIMES e2=e        {App(e1,Mult,e2)}
   | e1=e PLUS e2=e         {App(e1,Plus,e2)}
   | LPARA e1=e RPARA       {e1}
   | i1=INT                 {Const(i1)}
```

Priorités entre opérateurs

- La déclarations de plusieurs opérateurs comme associatifs à gauche ou à droite a une deuxième conséquence : dans une séquence de déclarations `%left` et `%right`, les opérateurs des lignes successives ont des priorités croissantes. On peut y mettre un opérateur non-associatif avec `%nonassoc`.
- Exemple :

```
%right @
%left $ #
%nonassoc &
```

l'opérateur & est le plus prioritaire, suivi par $ et # qui ont la même priorité, suivi par @ qui est le moins prioritaire.

Associativité et Priorités en menhir

```
%token <int> INT
%token DOLLAR SHARP AMP AT EOL PARG PARD
%start <Syntax.t> s
%right AT
%left DOLLAR SHARP
%nonassoc AMP
%{ open Syntax %}
%%
s: e1=e EOL                {e1}
e:   e1=e AT e2=e          {App(e1,At,e2)}
   | e1=e DOLLAR e2=e      {App(e1,Dollar,e2)}
   | e1=e SHARP e2=e       {App(e1,Sharp,e2)}
   | e1=e AMP e2=e         {App(e1,Amp,e2)}
   | PARG e1=e PARD        {e1}
   | i1=INT                {Const(i1)}
```

Comment menhir arbitre des conflits (1)

- On pourrait imaginer que menhir fait internement une transformation de la grammaire, un peu comme nous l'avons fait dans ce cours à la main.
- En vérité, les shift-reduce conflits sont arbitrés directement :
- D'abord, les niveau de priorité des jetons impliquent aussi des niveaux de priorités des règles : la priorité d'une règle $N \rightarrow \alpha$ est définie comme la priorité de son dernier jeton (celui qui est plus à droite dans α).
- Par exemple, la priorité de la règle $e \rightarrow e\ PLUS\ e$ est la même que la priorité du jeton PLUS.

Comment menhir arbitre des conflits (2)

- Considérons un état de l'automate qui contient deux items LR(1) : avec un shift-reduce conflit entre *shift* $\square$ et *reduce* $N \to \alpha$:
  1. Si $\square$ prioritaire sur $N \to \alpha$: shift !
  2. Si $N \to \alpha$ prioritaire sur $\square$: reduce !
  3. Si les deux ont la même priorité :
     - 3.1 Si $\square$ associatif à droite : shift !
     - 3.2 Si $\square$ associatif à gauche : reduce !
     - 3.3 Si $\square$ non-associatif : utilisation associative va déclencher une erreur de syntaxe !
- Sinon : Conflit pas résolu.

Retour à l'exemple du *dangling else*

- Arbitrer le conflit shift/reduce dans
  1. [expr → `IF` expr `THEN` expr ., {THEN, EOL, ELSE}]
  2. [expr → `IF` expr `THEN` expr . `ELSE` expr, {THEN, EOL, ELSE}]
- Normalement on souhaite que

  `IF ID THEN IF ID THEN ID ELSE ID`

  soit compris comme

  `IF ID THEN ( IF ID THEN ID ELSE ID )`
- Il faut donner à `ELSE` la priorité devant `THEN`.

Arbitrer entre THEN et ELSE

```
%token IF THEN ELSE ID EOL
%start <Syntax.expression> s
%nonassoc THEN
%nonassoc ELSE
%%

s: expr EOL                              {()}
expr: | ID                               {()}
   | IF expr THEN expr                   {()}
   | IF expr THEN expr ELSE expr         {()}
```

Utiliser la bibliothèque standard de menhir

- La bibliothèque standard de menhir fournit des raccourcis utiles pour écrire des règles de la grammaire, par exemple les opérateurs +, * et ? connus des expressions régulières, et plus (voir la doc complète).
- Si le non-terminal e produit des valeurs de type t, alors
  - e+ et e* produisent des valeurs de type t list ;
  - e? produit des valeurs de type t option.

Exemple d'utilisation de * et + en menhir

```
(* Syntaxe abstraite des expressions arithmétiques *)

type op = Plus | Mult
type t = Const of int | App of op * t list

let eval = function Plus -> (+) | Mult -> ( * )
let neutre = function Plus -> 0 | Mult -> 1

let rec calc = function
  | Const(i) -> i
  | App(o,l) -> List.fold_left
                  (fun a x -> (eval o) a (calc x))
                  (neutre o) l
```

Exemple d'utilisation de * et + en menhir

```
%token <int> INT
%token TIMES PLUS LPARA RPARA EOL
%start <Syntax.t> s
%{ open Syntax %}
%%

s: e1=e EOL                          {e1}
e:  TIMES LPARA l1=e+ RPARA          {App(Mult,l1)}
  | PLUS LPARA l1=e* RPARA           {App(Plus,l1)}
  | i1=INT                           {Const(i1)}
```

Implémentation dans la librairie menhir

Utilisation d'un mécanisme plus général : des *non terminaux paramétrés* (voir la documentation pour plus de détail).

```
separated_nonempty_list(separator, X):
| x = X
      { [ x ] }
| x = X separator
  xs = separated_nonempty_list(separator, X)
      { x :: xs }
```