

Langages de Programmation, Interprétation, Compilation

Christine Paulin

`Christine.Paulin@universite-paris-saclay.fr`

Département Informatique, Faculté des Sciences d'Orsay, Université Paris-Saclay

LDD3 Informatique, Mathématiques - Magistère Informatique

2025–26

1 Outillage pour l'analyse syntaxique

- Génération d'analyseurs lexicaux avec ocamllex
- Génération d'analyseurs syntaxiques avec menhir
- Conflits et priorités
- Analyse syntaxique de IMP
- Utilisation de LEX à d'autres fins que l'analyse lexicale

- Automatisation de la génération d'un analyseur lexical
- Automatisation de la génération d'un analyseur syntaxique
- Utilisation de la théorie des cours précédents
 - Construction des automates
 - Fonction d'analyse générique en fonction de l'automate
 - Intégration d'actions arbitraires
- Exemples de compilateurs (qui servent à construire d'autres compilateurs) :
 - traduction d'un langage dédié (expressions régulières, règles grammaticales) vers un langage général
 - des conventions lexicales et syntaxiques. . .
 - initié pour C avec lex et yacc
 - existe pour la plupart des langages

1 Outillage pour l'analyse syntaxique

- Génération d'analyseurs lexicaux avec ocamllex
- Génération d'analyseurs syntaxiques avec menhir
- Conflits et priorités
- Analyse syntaxique de IMP
- Utilisation de LEX à d'autres fins que l'analyse lexicale

- Automatiser la reconnaissance de lexèmes à partir d'expressions régulières et d'actions associées
- outils de la famille LEX : ocamllex pour Caml
- un fichier .mll contient l'ensemble des expressions régulières à reconnaître et les traitements associés
- l'utilitaire ocamllex traduit ce fichier en un programme caml

- Le cœur du fichier .mll : des expressions régulières à reconnaître et des traitements associés,

```
| expression_régulière_1  
{ traitement_1 }
```

```
| expression_régulière_2  
{ traitement_2 }
```

```
| ...
```

- s'apparente à la définition d'une fonction par cas :
- traduit en un automate et des fonctions de reconnaissance.

Prélude d'un fichier .mll

- Le fichier .mll contient un prélude et un épilogue
 - code caml arbitraire copié au début ou à la fin du fichier .ml produit.
- Prélude : zone entre accolades au début du fichier contenant
 - importations de modules

```
{ open Lexing  
  open Printf
```

- définitions de types et d'exceptions

```
type token =  
  | IDENT of string  
  | INT   of int  
  | ...  
exception Eof
```

- déclarations de variables globales

```
let lines = ref 0  
let file = Sys.argv.(1)  
let cout = open_out (file ^ ".doc")
```

- définitions de fonctions auxiliaires

```
let print s = fprintf cout s }
```

Définition d'expressions régulières auxiliaires

- Règles de reconnaissance : placées immédiatement après le prélude
- Syntaxe : spécifique à ocamllex
- Avant les règles : raccourcis pour des expressions régulières

let nom = expression_régulière

Syntaxe ocamllex des expressions régulières

<code>_</code>	n'importe quel caractère
<code>'a'</code>	caractère spécifique
<code>"abc"</code>	chaîne de caractères spécifique
<code>[...]</code>	alternative parmi un ensemble de caractères
<code>[^...]</code>	alternative parmi le complément d'un ensemble de caractères
<code>r₁ r₂</code>	alternative
<code>r₁ r₂</code>	concaténation
<code>r*</code>	étoile (répétition de <code>r</code> , éventuellement vide)
<code>r+</code>	répétition non vide
<code>r?</code>	présence optionnelle de <code>r</code>
<code>eof</code>	fin de l'entrée

L'usage des conventions caml pour caractères et chaînes permet de distinguer les caractères de l'alphabet `'*'` des opérations sur les expressions régulières.

- Priorités :
 - priorité la plus forte sur les opérateurs postfixes $?+*$
 - ensuite la concaténation
 - enfin l'alternative
- Exemple `'a' | 'b'* c` représente `'a' | (('b'* c)`
- Utilisation de parenthèses si besoin

Définition d'un ensemble de caractères

- `[ . . . ]` alternative parmi un ensemble de caractères
- `[ ^ . . . ]` exclusion d'un ensemble de caractères
- énumérer les caractères en les séparant par une espace : `[ ' a' ' b' ' c' ]`
- sélectionner toute une plage en utilisant un tiret : `[ ' a' - ' c' ]`
- combiner les deux : `[ ' a' - ' z' ' _' ]`

Exemples

reconnaitre un chiffre

```
let digit = ['0' - '9']
```

reconnaitre une lettre, minuscule ou majuscule

```
let alpha = ['a' - 'z' 'A' - 'Z']
```

identifiant caml : lettre suivie d'une suite de chiffres, lettres et '_'.

```
let ident = alpha (digit | alpha | '_')*
```

partie décimale d'un nombre : un point et une suite quelconque de chiffres

```
let decimals = '.' digit*
```

exposant : lettre e, suivie d'un nombre entier positif ou négatif le signe de l'exposant est optionnel.

```
let exponent = ['e' 'E'] ['+' '-']? digit+
```

nombre flottant : partie décimale et/ou un exposant

```
let fnumber = digit+ (decimals | decimals? exponent)
```

Définition d'une fonction d'analyse

- On définit les règles de reconnaissance et les traitements associés.
- Les règles de reconnaissance sont regroupées sous un nom de fonction introduit par :
rule nom_de_la_fonction = parse
- le fichier .ml produit par ocamllex, définira une fonction du nom correspondant, de type Lexing.lexbuf -> res
- Lexing.lexbuf fait référence à la structure lexbuf définie dans le module Lexing, qui décrit une entrée en cours de lecture,
- res est le type de retour des traitements réalisés.
- il est possible de définir plusieurs analyseurs de manière récursive
let rec f x = ... **and** g y = ...

Exemple

- fonction de parcours de texte
 - on recopie les commentaires introduits par `/**` et qui se termine par `*/` dans un fichier
 - on reconnaît des lexèmes qu'on affiche sur la sortie standard
- utilisation de deux fonctions
 - `scan_token` : lit le texte, cherche à reconnaître un lexème, lorsqu'un début de commentaire est trouvé, appelle `scan_text` pour traiter le commentaire puis cherche le lexème suivant
 - `scan_text` : copie le commentaire dans le fichier, compte le nombre de lignes de commentaires, se termine à la fin du commentaire
- Le fichier `.ml` généré par `ocamllex` contient la définition de fonction `scan_text` : `Lexing.lexbuf -> unit`.

Voir fichier `lexer1.mll`

Compilation :

- `ocamllex lexer1.mll`
- `ocamlopt lexer1.ml -o scan`
- `./scan fichier`

- utilitaires du module Lexing :
 - variable prédéfinie `lexbuf` : `lexbuf` : l'entrée en cours de lecture
 - l'entrée est "consommée" au fur et à mesure de la lecture
 - chaîne reconnue : `lexeme: lexbuf -> string` appliquée à `lexbuf` : `lexbuf`
 - on peut aussi utiliser la notation **as** dans l'expression régulière pour nommer la chaîne correspondant à la reconnaissance d'une (sous)expression régulière

```
| ident as s { IDENT s }
```

- l'entrée reconnue est une chaîne de caractères qui devra être traduite en entier, flottant. . .
- on peut appliquer récursivement la fonction `scan_text`

Priorités des règles

- l'analyse cherche toujours à reconnaître une chaîne la plus longue possible
- si plusieurs règles sont susceptibles de reconnaître un préfixe de la chaîne analysée, on applique la règle qui reconnaît le plus long préfixe.
- exemple `print_int 3`
 - la règle du mot-clé "print" reconnaît le mot `print`
 - la règle des identifiants reconnaît le mot `print_int`.
- si deux règles reconnaissent des mots de même longueur, la première est choisie
- exemple `print 3`
 - reconnaît le mot-clé "print" qui apparaît en premier
- même phénomène avec la reconnaissance des entiers ou des flottants

Épilogue d'un fichier .ml1

- code caml arbitraire qui sera intégré tel quel au fichier .ml produit
- délimité par des accolades
- fait référence à ce qui a été défini dans le prélude, et aux fonctions de reconnaissance définies dans la partie principale
- si l'analyseur constitue le programme principal, on peut mettre l'équivalent d'une fonction main
- si l'analyseur lexical est utilisé dans un compilateur, c'est l'analyseur syntaxique qui demandera les lexèmes un à un à mesure des besoins de son analyse.

- la reconnaissance des mot-clés du langage pourrait se faire avec une règle par mot clé
- les mots clés appartiennent au langage des identificateurs
- les distinguer engendre un automate inutilement gros
- optimisation
 - une seule règle qui reconnaît le langage des identifiants
 - tester dans l'action si la séquence reconnue appartient est un mot-clé pour renvoyer le bon lexème.
- même chose si on veut des identifiants insensibles à la casse : reconnaissance d'une classe générale puis "normalisation" dans l'action

Différents niveaux de lecture

- Distinction entre syntaxe abstraite et la syntaxe concrète, dans le langage cible, mais aussi dans ocamllex
- 4 niveaux de lecture
 - La valeur sémantique de ce que l'on veut reconnaître (un entier, une chaîne, . . .) qui apparaîtra dans l'arbre de syntaxe abstraite et qui sert au calcul
 - la manière dont le langage traité demande d'écrire cette valeur (flottant, chaînes avec délimiteurs, caractères d'échappement,...)
 - l'expression régulière qui traduit la syntaxe concrète
 - la manière dont l'outil de génération d'analyseurs demande d'écrire l'expression régulière
- Exemple
 - valeur sémantique : le caractère ASCII de code 10 qui représente le retour à la ligne
 - syntaxe concrète Caml : `\n`
 - expression régulière : le caractère `\` suivi du caractère `n`
 - expression régulière ocamllex : `"\\ 'n'`

1 Outillage pour l'analyse syntaxique

- Génération d'analyseurs lexicaux avec ocamllex
- **Génération d'analyseurs syntaxiques avec menhir**
- Conflits et priorités
- Analyse syntaxique de IMP
- Utilisation de LEX à d'autres fins que l'analyse lexicale

- Construire les automates d'analyse ascendante est complexe à faire à la main, mais peut être automatisé
- outils de la famille YACC (Yet Another Compiler Compiler),
- en Caml ocaml yacc et menhir.
- on décrit dans un fichier .mly les règles de la grammaire à reconnaître, et les traitements associés (construire l'AST)
- menhir traduit ce fichier en un programme Caml réalisant une analyse ascendante d'un texte en suivant la grammaire fournie
- le programme Caml obtenu prend en entrée le texte à analyser, mais aussi une fonction d'analyse lexicale fournissant à la demande le prochain lexème (générée par ocamllex)
- l'outil menhir émet un avertissement et un diagnostic en cas de conflit dans l'analyse.

Structure d'un fichier `.mly`

- Le cœur du fichier `.mly` est constitué des règles de la grammaire à reconnaître et des traitements associés
- Le fichier commence et termine par un prélude et un épilogue contenant des fragments de code Caml intégrés respectivement au début et à la fin du fichier `.ml` produit
- ces zones sont délimitées par `%{` et `%}`

Prélude d'un fichier .mly

- zone qui définit le contexte et les éléments utilisés dans les actions.



les types de retour des fonctions d'analyse doivent être définis dans des modules séparés.

- module ast.ml

```
type expression =  
    | Cst of int  
    | Add of expression * expression  
    | Mul of expression * expression  
type program =  
    { code : expression }
```

- prélude du fichier .mly

```
%{  
    open Ast  
}%
```

Déclaration des symboles de la grammaire

- Les lexèmes manipulés (symboles terminaux de la grammaire), sont déclarés en tête de la partie principale

```
% token nom_du_lexeme
```

pour les lexèmes ordinaires sans contenu et

```
% token <type_du_contenu> nom_du_lexeme
```

pour les lexèmes portant une valeur.

- Le symbole non terminal de départ de la grammaire est déclaré avec

```
% start nom_du_symbole
```
- Cette déclaration est obligatoirement associée à une déclaration de type, indiquant le type de la valeur produite par l'analyse syntaxique

```
% type <type_du_resultat> nom_du_symbole
```
- La déclaration des types des autres symboles non terminaux est optionnelle.
- On peut déclarer plusieurs lexèmes par ligne.

Type et fonctions engendrées

Le programme Caml généré par menhir à partir de cette grammaire exporte

- un type token des non terminaux
- une fonction portant le nom du symbole de départ, et de type
`(Lexing.lexbuf -> token) -> Lexing.lexbuf -> type_du_resultat`
- Cette fonction prend en paramètre une fonction d'analyse lexicale qui fournit le prochain lexème, ainsi que l'entrée à lire (utilisée pour l'analyse lexicale).
- Le résultat a le type déclaré pour le symbole de départ

Exemple

Entête du fichier `parser1.mly`

```
%{  
    open Ast  
%}  
%token <int> INT  
%token LPAR RPAR PLUS STAR  
%token EOF  
%type <Ast.program> prog  
%type <Ast.expression> expr (* optionnel *)  
%start prog
```

Exemple

Fichier `parser1.mli` engendré

type token =

- | STAR
- | RPAR
- | PLUS
- | LPAR
- | INT **of** (int)
- | EOF

exception Error

val prog: (Lexing.lexbuf -> token) -> Lexing.lexbuf
 -> (Ast.program)

Définition des règles et des traitements

- fin de la déclaration des symboles et début des règles : une ligne comportant uniquement

`%%`

- chaque symbole non terminal est introduit par :

`nom_du_symbole :`

suivie des règles associées

- une règle contient un motif, puis entre accolades un traitement associé (calculé lors de la réduction de la règle) de cette règle.
- le type de retour correspond au type déclaré pour le symbole non terminal en cours de définition
- on fait référence aux valeurs correspondant aux symboles de la production avec la notation (ocamlyacc et menhir)

`$numero_du_symbole`

La numérotation prend en compte tous les symboles de la production, qu'ils soient terminaux ou non terminaux, et progresse de gauche à droite (en commençant par 1).

- menhir propose une notation plus lisible en associant un nom de variable à un symbole qui représente sa valeur dans l'action

Exemple

règle $\text{expr} \prec \text{LPAR expr RPAR}$

- Notation yacc

```
expr :  
| LPAR e=expr RPAR { $2 }  
;
```

- notation menhir

```
expr :  
| LPAR e=expr RPAR { e }
```

- on récupère ainsi les valeurs associées aux non-terminaux construites par l'analyse syntaxique mais aussi les valeurs des terminaux données par l'analyse lexicale

```
| i=INT { Cst i }
```

- Les éléments d'une règle peuvent optionnellement être séparés par des point-virgules pour faciliter la lecture.

```
| e1=expr; PLUS; e2=expr { Add(e1, e2) }  
| expr STAR expr { Mul($1, $3) }  
;
```

Épilogue d'un fichier .mly.

- La zone de description des règles termine comme elle a commencé par une ligne contenant uniquement les symboles %%.
- Le code de l'épilogue, est délimité par %{ et %}
- L'épilogue est du code caml copié à la fin du fichier, il peut faire référence à la fonction produite pour le symbole de départ.

Et maintenant...

Avec la grammaire proposée :

```
menhir parser1.mly
```

```
Warning: 2 states have shift/reduce conflicts.
```

```
Warning: 4 shift/reduce conflicts were arbitrarily resolved
```

Plusieurs grammaires alternatives pour les expressions arithmétiques :

- Une grammaire avec niveaux de priorités, SLR(1) (donc LR(1)).

$$\begin{array}{lcl} E & ::= & E + M \\ & | & M \\ M & ::= & M * A \\ & | & A \\ A & ::= & n \\ & | & (E) \end{array}$$

- Une grammaire LL(1)

$$\begin{array}{lll} E & ::= & M E' \\ E' & ::= & + E \\ & | & \varepsilon \\ M & ::= & A M' \\ M' & ::= & * M \\ & | & \varepsilon \\ A & ::= & n \\ & | & (E) \end{array}$$

écriture directe d'un analyseur récursif descendant, est aussi LR(1),
comment reconstruire l'ast ??

Détection d'un conflit

Garder la grammaire naturelle (simplifiée) des expressions arithmétiques :

```
%token INT PLUS LPAR RPAR EOF
```

```
%start prog
```

```
%type <unit> prog
```

```
%%
```

```
prog:
```

```
| expr EOF {}
```

```
expr:
```

```
| INT {}
```

```
| expr PLUS expr {}
```

```
| LPAR expr RPAR {}
```

Bilan donné par menhir :

```
menhir -v --infer parser2.mly
```

```
Warning : one state has shift/reduce conflicts.
```

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

On a dans le fichier `parser2.automaton` produit par `menhir` (option `-v`)

```
State 5:
## Known stack suffix:
## expr PLUS
## LR(1) items:
expr -> expr PLUS . expr [ RPAR PLUS EOF ]
## Transitions:
-- On LPAR shift to state 1
-- On INT shift to state 2
-- On expr shift to state 6
## Reductions:
```

Exemple

état comportant un conflit.

State 6:

Known stack suffix:

expr PLUS expr

LR(1) items:

expr -> expr . PLUS expr [RPAR PLUS EOF]

expr -> expr PLUS expr . [RPAR PLUS EOF]

Transitions:

-- On PLUS shift to state 5

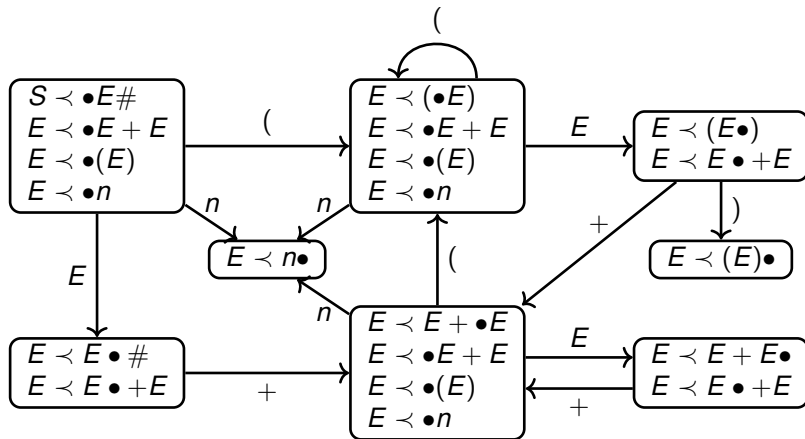
Reductions:

-- On RPAR PLUS EOF

-- reduce production expr -> expr PLUS expr

** Conflict on PLUS

Automate LR(0) associé



Analyse du conflit

Fichier . conflicts détaille les conflits

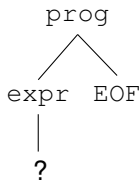
```
** Conflict (shift/reduce) in state 6.  
** Token involved: PLUS  
** This state is reached from prog after reading:
```

```
expr PLUS expr
```

arbres de dérivation correspondant à l'un ou l'autre choix :

```
** The derivations that appear below have the following common  
** factor: (The question mark symbol (?) represents the spot whe  
** the derivations begin to differ.)
```

```
prog  
expr EOF  
(?)
```



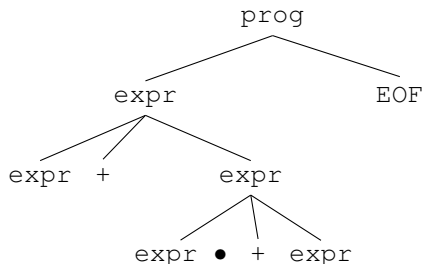
Analyse d'un conflit

Chaque action (progression ou réduction), correspond à un sous-arbre
Arbre pour la progression

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

```
expr PLUS expr
      expr . PLUS expr
```

Arbre correspondant :



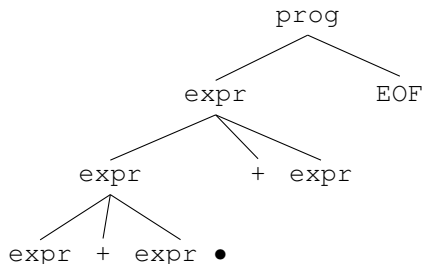
Analyse d'un conflit

Arbre pour la réduction.

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

```
expr PLUS expr // lookahead token appears  
expr PLUS expr .
```

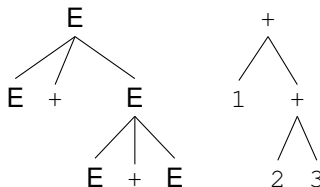
Arbre correspondant :



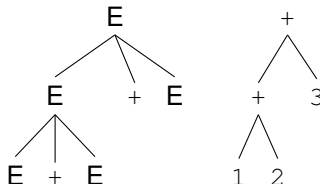
Comment choisir ?

Choix possibles pour l'entrée $1+2+3$

- progresser avec le symbole $+$, parenthésage $1+(2+3)$.



- réduire avec la règle $E \rightarrow E+E$, parenthésage $(1+2)+3$



- déterminer laquelle de ces deux interprétations est « la bonne », et indiquer en conséquence à l'outil s'il doit choisir de progresser ou de réduire.
- on choisit un parenthésage implicite à gauche, c'est-à-dire favoriser $(1+2)+3$.
- on déclare l'opérateur PLUS comme « associatif à gauche ».
- directive après la déclaration des symboles terminaux

`%left PLUS`

- pour un opérateur associatif à droite, on utilise la directive `%right`.
- pour un opérateur non-associatif (par exemple une comparaison), on utilise la directive `%nonassoc`.

Conflits avec plusieurs symboles

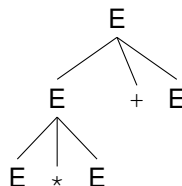
Avec la multiplication $*$ (symbole terminal STAR) on a trois nouveaux conflits :

- réduire `expr STAR expr` ou progresser avec STAR,
- réduire `expr STAR expr` ou progresser avec PLUS,
- réduire `expr PLUS expr` ou progresser avec STAR.
- premier cas : associativité de la multiplication
- autres cas : priorités relatives données aux opérations d'addition et de multiplication.

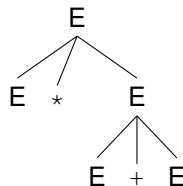
Conflits avec plusieurs symboles

Si on a reconnu `expr STAR expr` est que le prochain symbole est `PLUS` on peut :

- réduire : interpréter $2*3+4$ comme $(2*3)+4$



- progresser : interpréter $2*3+4$ comme $2*(3+4)$

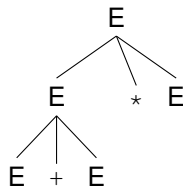


conventions mathématiques usuelles : favoriser la réduction.

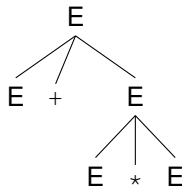
Analyse d'un conflit

Si on a reconnu `expr PLUS expr`, prochain symbole `STAR` :

- réduire : interpréter $2+3*4$ comme $(2+3)*4$,



- progresser : interpréter $2+3*4$ comme $2+(3*4)$,



conventions mathématiques usuelles : favoriser la progression.

Déclaration de priorités avec menhir

- Point commun à ces deux conflits : l'opérateur de multiplication doit être plus prioritaire que l'opérateur d'addition.
- on place les informations d'associativité de PLUS et STAR sur deux lignes différentes.
- informations d'associativité par ordre de priorité, en commençant par les opérateurs les moins prioritaires.

```
% l e f t  PLUS
```

```
% l e f t  STAR
```

Déclaration de priorités

- Les déclarations concernent toutes les utilisations des symboles concernés.
- un même symbole peut apparaître dans plusieurs règles, avec des priorités différentes.
- Le symbole $-$ par exemple peut apparaître dans deux situations :
 - en tant qu'opérateur binaire de soustraction, $1 - 2$,
 - en tant qu'opérateur unaire, pour l'« opposé », $- 1$.

Exemple

- expression $-2 * 3 - 4 * 5$
- parenthésage implicite conventionnel : $((-2)*3) - (4*5)$.
- le symbole $-$ unaire est plus prioritaire que la multiplication
- le symbole $-$ binaire est moins prioritaire que la multiplication
- on va introduire

lorsqu'il représente une opération unaire, et moins prioritaire que la multiplication lorsqu'il représente une opération binaire.

Priorités multiples

- on peut introduire dans menhir deux symboles :
 - un symbole terminal normal MINUS désignant –
 - un symbole terminal fantôme U_MINUS utilisé seulement pour marquer la priorité du – unaire.
- on force la priorité de la règle du – unaire à être celle de U_MINUS

```
%token PLUS STAR MINUS U_MINUS
```

```
%left PLUS MINUS
```

```
%left STAR
```

```
%nonassoc U_MINUS
```

```
%%
```

```
expr :
```

```
| e1=expr STAR e2=expr { Mul(e1, e2) }
```

```
| e1=expr MINUS e2=expr { Sub(e1, e2) }
```

```
| MINUS e=expr { Opp(e) } %prec U_MINUS
```

Priorités : fonctionnement

- chaque opération a une priorité :
 - progression sur un symbole $a \in T$: la priorité de a ,
 - réduction d'une règle $X \prec \beta$, la priorité du symbole terminal le plus à droite dans la séquence β ou celle forcée par %prec
- La résolution est séparée en deux cas :
 - 1 si les priorités sont différentes : on réalise l'opération la plus prioritaire,
 - 2 sinon on utilise l'associativité :
 - réduction pour des opérateurs associatifs à gauche
 - progression pour des opérateurs associatifs à droite
 - erreur si non associatif
- deux opérateurs ont la même priorité s'ils sont sur la même ligne. Ils ont aussi la même associativité (indication %left ou %right).

- Utiliser les notions de priorité pour régler les conflits, et même des ambiguïtés de la grammaire, sans changer la grammaire elle-même.
- Choisir entre les différentes possibilités : analyser les différents arbres de dérivation possibles pour trouver ceux qui correspondent à l'interprétation voulue.
- Positionner les priorités correctement.

1 Outillage pour l'analyse syntaxique

- Génération d'analyseurs lexicaux avec ocamllex
- Génération d'analyseurs syntaxiques avec menhir
- Conflits et priorités
- **Analyse syntaxique de IMP**
- Utilisation de LEX à d'autres fins que l'analyse lexicale

Analyse syntaxique de IMP

Utilisation conjointe de `ocamllex` et `menhir` pour construire un nouvel analyseur syntaxique pour le langage IMP.

On organise ce programme en quatre parties :

- un module `Imp` (fichier `imp.ml`) définit l'AST et d'éventuelles fonctions auxiliaires de manipulation de la syntaxe abstraite,
- un module `Impparser` réalisé avec `menhir` (fichier `impparser.mly`) définit les lexèmes et l'analyseur syntaxique,
- un module `Implexer` réalisé avec `ocamllex` (fichier `implexer.mll`) définit l'analyseur lexical,
- un module principal `Impc` (fichier `impc.ml`) définit le programme principal, qui analyse un fichier fourni en entrée.

Syntaxe concrète du chapitre 3, avec quelques variations

- liste des opérateurs
- commentaires
- branche secondaire **else** optionnelle

Définition de l'AST (fichier `imp.ml`).

```
type bop = Add | Sub | Mul | Lt | Le
type expr =
  | Cst of int
  | Var of string
  | Bop of bop * expr * expr
type instr =
  | Print of expr
  | Set of string * expr
  | While of expr * instr list
  | If of expr * instr list * instr list
type prog = instr list
```

Définition de l'analyseur syntaxique avec menhir (fichier impparser.mly).

```
%{  
  open Imp  
  open Lexing  
%}  
%token <int> INT  
%token <string> IDENT  
%token PLUS MINUS STAR LT LE  
%token LPAR RPAR BEGIN END SEMI  
%token PRINT SET IF ELSE WHILE  
%token EOF
```

Non terminaux : P (programme complet), I (instruction) et E (expression)

```
 $P$  ::=  $I^* \#$ 

 $I$  ::=  $x := E$  ;
      | while (  $E$  ) {  $I^*$  }
      | if (  $E$  ) {  $I^*$  } [else {  $I^*$  }]?
      | print (  $E$  ) ;

 $E$  ::=  $n$ 
      |  $x$ 
      | (  $E$  )
      |  $E \ o \ E$           avec  $o \in \{ +, -, *, <, <= \}$ 
```

- $\#$ symbole terminal spécial de fin de fichier,
- n une constante entière,
- x un identifiant de variable,
- I^* une succession d'un nombre quelconque d'instructions,
- $[\text{else } \{ I^* \}]?$ branche **else** optionnelle
- conventions de priorité usuelles des opérateurs

Déclaration de la grammaire

Symbole de départ prog et son type.

```
%start prog  
%type <Imp.prog> prog  
%%
```

reconnaître une séquence d'instructions I^* : symbole seq

$$S ::= \varepsilon \mid I S$$

traduction menhir

```
seq :  
| i=instr s=seq { i :: s }  
| (* empty *)   { [] }  
;
```

À noter : la traduction d'une règle $S \rightarrow \varepsilon$: il suffit de ne rien écrire dans le motif à réduire !

Déclaration de la grammaire

instr :

```
| PRINT LPAR e=expr RPAR SEMI      { Print(e)      }  
| id=IDENT SET e=expr SEMI         { Set(id, e)    }  
| WHILE LPAR e=expr RPAR BEGIN s=seq END { While(e, s)  }  
| IF LPAR e=expr RPAR BEGIN s=seq END   { If(e, s, []) }  
| IF LPAR e=expr RPAR BEGIN s1=seq END  
    ELSE BEGIN s2=seq END             { If(e, s1, s2) }  
;
```

prog :

```
| s=seq EOF { s }
```

- rapporter à l'utilisateur des informations en cas d'erreur de syntaxe
- on ajoute à prog une règle d'erreur, qui récupère la position à laquelle l'analyse a échoué avec `$startpos` (une valeur spéciale de menhir)
- traduit cette position en un numéro de ligne et un numéro de colonne (voir le module Lexing pour le format des positions).

```
| error
  { let pos = $startpos in
    let message = Printf.sprintf "echec_à_la_position_%d,_%d"
                                pos.pos_lnum (pos.pos_cnum - pos.pos_bol)
    in
    failwith message }
;
```

Construction de l'arbre

symbole supplémentaire bop pour factoriser les différentes opérations binaires.

```
expr :  
| n=INT          { Cst(n) }  
| id=IDENT       { Var(id) }  
| LPAR e=expr RPAR { e }  
| e1=expr op=bop e2=expr { Bop(op, e1, e2) }  
;
```

définition des opérateurs binaires.

```
%inline bop:  
| PLUS { Add }  
| MINUS { Sub }  
| STAR { Mul }  
| LT { Lt }  
| LE { Le }  
;
```

À noter la directive `%inline`, qui demande à menhir d'expanser le symbole non terminal `bop` à chaque endroit où il apparaîtrait, c'est-à-dire de remplacer la règle

```
| e1=expr op=bop e2=expr  { Bop(op, e1, e2) }
```

par les cinq règles suivantes.

```
| e1=expr PLUS  e2=expr  { Bop(Add, e1, e2) }  
| e1=expr MINUS e2=expr  { Bop(Sub,  e1, e2) }  
| e1=expr STAR  e2=expr  { Bop(Mul, e1, e2) }  
| e1=expr LT    e2=expr  { Bop(Lt,  e1, e2) }  
| e1=expr LE    e2=expr  { Bop(Le,  e1, e2) }
```

```
%nonassoc LT LE  
%left PLUS MINUS  
%left STAR
```

- `menhir` fournit une primitive `list` pour la reconnaissance d'une séquence, qui renvoie une liste `caml`.
- On peut se passer complètement du symbole `seq`

```
prog :  
  | s=list(instr) EOF { s }
```

On a également sur le même modèle : `nonempty_list` et `separated_list`.

Quelques bonus de `menhir`

- `menhir` fournit une primitive `option`
- renvoie une option `caml` (`None` si l'élément optionnel n'est pas présent, `Some s` sinon).
- variante `loption`, lorsque l'élément optionnel est une liste : renvoie directement la liste reconnue, ou `[]` si la liste optionnelle n'est pas présente.
- Exemple avec une nouvelle règle `else_` et une seule règle pour **if**.

```
else_  
| ELSE BEGIN s=list(instr) END { s }  
;  
instr:  
| IF LPAR c=expr RPAR BEGIN s1=list(instr) END  
s2=loption(else_) { If(c, s1, s2) }
```

En développant

```
instr:  
| IF LPAR c=expr RPAR BEGIN s1=list(instr) END  
s2=loption(ELSE BEGIN s=list(instr) END {s}) { If(c, s1, s2)
```

- On réalise un analyseur lexical avec ocamllex (fichier implexer.mll).
- le prélude charge le module Impparser, qui définit les lexèmes que l'analyse lexicale doit produire.
- on utilise une table des mots clés et des lexèmes associés
- La fonction `new_line` est définie dans la bibliothèque Lexing. On l'appelle à chaque retour à la ligne pour obtenir des messages d'erreurs identifiant la ligne du fichier concernée.

Programme principal

Le module principal (fichier `impc.ml`) récupère un nom de fichier en ligne de commande, et analyse ce fichier.

```
let () =  
  let file = Sys.argv.(1) in  
  let c = open_in file in  
  let lexbuf = Lexing.from_channel c in  
  let ast = Impparser.prog Implexer.token lexbuf in  
  close_in c;  
  ignore(ast);  
  exit 0
```

1 Outillage pour l'analyse syntaxique

- Génération d'analyseurs lexicaux avec ocamllex
- Génération d'analyseurs syntaxiques avec menhir
- Conflits et priorités
- Analyse syntaxique de IMP
- Utilisation de LEX à d'autres fins que l'analyse lexicale

les outils de la famille LEX sont utiles pour réaliser tout programme analysant un texte (chaîne de caractères, fichier, flux...) sur la base d'expressions régulières, ou transformant un texte par une série de modifications locales relativement simples.

Nettoyage d'un texte

Pour récupérer un texte sur l'entrée standard et produire sur la sortie standard le même texte dans lequel les lignes vides consécutives sont ignorées (on en garde une seule).

Fichier mbl.mll :

```
rule scan = parse
| '\n' '\n'+ { print_string "\n\n"; scan lexbuf }
| _ as c      { print_char c; scan lexbuf }
| eof        { () }

{ let _ = scan (Lexing.from_channel stdin) }
```

on fabrique l'exécutable :

```
# ocamllex mbl.mll
# ocamlopt -o mbl mbl.ml
```

on l'utilise pour lire un fichier infile et écrire le résultat dans un fichier outfile avec la ligne de commande

```
# ./mbl < infile > outfile
```

Programme qui prend en paramètres sur la ligne de commande un mot (suite de caractères alphabétiques) et un nom de fichier, et affiche le nombre d'occurrences du mot dans le fichier.

```
{
  let word = Sys.argv.(1)
  let count = ref 0
}

rule scan = parse
| ['a'-'z' 'A'-'Z']+ as w { if word = w then incr count;
                             scan lexbuf }
| _ { scan lexbuf }
| eof { () }

{
  let () = scan (Lexing.from_channel (open_in Sys.argv.(2)))
  let () = Printf.printf "%d_occurrence(s)\n" !count
}
```

A retenir

- Principes de base d'un analyseur lexical engendré par ocamllex :
 - expressions régulières et actions (récupération de la chaîne reconnue)
 - principe de la reconnaissance du préfixe le plus long
 - méthodes d'analyse des principales classes de lexèmes (expressions régulières associées)
- Principes de base d'un analyseur syntaxique engendré par menhir :
 - déclaration des terminaux, non-terminaux et valeurs associées
 - construction des valeurs dans les actions
 - interaction entre l'analyseur syntaxique et lexical

Savoir faire

- Lire et écrire un fichier ocamllex
- Lire et écrire un fichier menhir
- Comprendre et résoudre les situations de conflit