

TP n°1

L'objectif de ce TP est de construire un petit projet permettant d'analyser lexicalement et syntaxiquement un fichier d'entrée.

1 Point de départ

Afin de réaliser ce TP, vous devez avoir installé le *compilateur* `ocamlc` sur votre machine, qui, dans les installations classiques, vient avec les outils `ocamllex` et `ocamlyacc`. Le projet est également prévu pour fonctionner sur un environnement simplement composé d'un shell (`bash` typiquement) et de l'outil `make`.

Le code initial est téléchargeable sur eprel sous la forme d'une archive `inter_tp1.zip`. Après avoir téléchargé et décompressé ce fichier, vous obtenez les fichiers suivants constituant le projet :

- `makefile` : le fichier de spécification pour l'outil `make`
- `ast.ml` : le fichier du module **Ast** (**syntaxe abstraite**)
- `parser.mly` : le fichier de spécification de l'analyseur syntaxique (**syntaxe concrète**)
- `lexer.mll` : le fichier de spécification de l'analyseur lexical
- `main.ml` : le fichier contenant le code du programme principal

1.1 Makefile et organisation du code

Le fichier `makefile` est constitué de deux parties :

1. Une partie customisable permettant de spécifier le nom de l'exécutable et la liste des fichiers constituant le projet ;

```
# Nom du fichier exécutable généré
EXEC=tp

# Liste des sources du projet, données dans l'ordre
SOURCES= \
    ast.ml      \
    parser.mly  \
    lexer.mll   \
    main.ml
```

2. Une partie générique à ne pas éditer permettant de spécifier les règles de compilation.

La compilation du projet se fait simplement à l'aide de la commande `make` dans le terminal.

```
$ make
ocamlyacc -v parser.mly
ocamllex -o lexer.ml lexer.mll
10 states, 270 transitions, table size 1140 bytes
```

```
ocamldep ast.ml parser.ml lexer.ml main.ml > depend
ocamlc -o ast.cmo -c ast.ml
ocamlc -o parser.cmo -c parser.ml
ocamlc -o lexer.cmo -c lexer.ml
ocamlc -o main.cmo -c main.ml
ocamlc -o tp ast.cmo parser.cmo lexer.cmo main.cmo
```

Pour supprimer l'ensemble des fichiers générés, on utilisera la commande `make clean`.

```
$ make clean
rm -rf tp parser.ml parser.mli parser.output lexer.ml ast.cmo \
    parser.cmo lexer.cmo main.cmo ast.cmi parser.cmi lexer.cmi main.cmi \
    depend
```

La compilation du projet produit un fichier exécutable nommé dans notre cas `tp` (variable `EXEC` dans `makefile`), qu'il convient de lancer comme tout autre programme.

```
$ ./tp
1 ;;
1
1 + 2 ;;
(1) + (2)
1 + 2 + 3 ;;
((1) + (2)) + (3)
Bye
```

Notre programme se présente comme une boucle interactive attendant en entrée standard une suite d'expressions arithmétiques terminées par un double point-virgules `;;`. Chaque expression est analysée et évaluée. Le programme affiche l'arbre de syntaxe reconnu. Pour quitter le programme, il suffit d'indiquer la *fin de fichier* sur l'entrée standard, par un `Ctrl+D` par exemple. Le code correspondant se trouve dans `main.ml`.

1.2 Syntaxe abstraite

La grammaire considérée dans le projet est la suivante (elle sera étendue par la suite).

$$\begin{array}{ll}
 E \rightarrow \underline{\text{int}} & \text{(constante entière)} \\
 | \quad E \pm E & \text{(addition)} \\
 | \quad \underline{(E)} & \text{(parenthésage)}
 \end{array}$$

Le fichier `ast.ml` contient la description de la syntaxe abstraite (**A**bstract **S**yntax **T**ree) associée à travers la définition du type `expr`.

```
type expr =
  | ECst of int
  | EAdd of expr * expr
```

Comme vu en cours, on remarque qu'un constructeur est donné pour chaque production de la grammaire, avec en paramètres les informations importantes.

— $E \rightarrow \underline{\text{int}}$: `ECst` avec un entier en paramètre,

- $E \rightarrow E \pm E$: `EAdd` avec en paramètres les sous-arbres correspondant aux 2 opérandes.

On remarque que la production correspondant au parenthésage $E \rightarrow (E)$ n'est pas conservée : les parenthèses n'ayant qu'une utilité syntaxique (celle de protéger un sous-calcul, comme dans « $(1 + 2) + 3$ ») et aucun effet calculatoire (la valeur de « $(1 + 2)$ » est la même que celle de « $1 + 2$ »), les parenthèses sont simplement ignorées dans la syntaxe abstraite.

Le fichier contient également la définition de la fonction `string_of_expr` permettant de générer une chaîne à partir d'un AST.

```
let rec string_of_expr e = match e with
| ECst i -> string_of_int i
| EAdd (e1, e2) ->
  Printf.sprintf "(%s) + (%s)"
    (string_of_expr e1) (string_of_expr e2)
```

1.3 Analyseur syntaxique

Le fichier `parser.mly` contient la spécification de l'analyseur syntaxique qui nous intéresse. Le fichier suit l'organisation présentée <https://caml.inria.fr/pub/docs/manual-ocaml/lex yacc.html#sec308>.

La grammaire spécifiée dans le fichier correspond à la suivante :

$$\begin{array}{ll}
 S & \rightarrow \text{;; } S \mid E \text{ ;;} \\
 E & \rightarrow \underline{\text{int}} \quad \text{(constante entière)} \\
 & \mid E \pm E \quad \text{(addition)} \\
 & \mid (E) \quad \text{(parenthésage)}
 \end{array}$$

qui embarque la grammaire précédente afin de gérer le terminateur `;;`.

Le fichier est structuré de la façon suivante :

- Un *header* permettant de charger le module `Ast` correspondant au fichier `ast.ml`.
- La déclaration des unités lexicales du point de vue token.
 - `%token <int> INT` correspondant à `int` attribué par un entier
 - `%token PLUS` pour `±`
 - `%token LPAR RPAR` pour `(` et `)` respectivement
 - `%token TERM` pour `;;`
- Les précédences pour la résolution des conflits.

Question 1. Supprimer la précedence `%left PLUS` pour voir apparaître le conflit après recompilation :

```
$ make
ocamlyacc -v parser.mly
1 shift/reduce conflict.
ocamldep ast.ml parser.ml lexer.ml main.ml > depend
ocamlc -o parser.cmo -c parser.ml
ocamlc -o lexer.cmo -c lexer.ml
ocamlc -o main.cmo -c main.ml
ocamlc -o tp ast.cmo parser.cmo lexer.cmo main.cmo
```

Question 2. Observer le fichier `parser.output` où le conflit est décrit.

```
13: shift/reduce conflict (shift 10, reduce 4) on PLUS
state 13
expr : expr . PLUS expr (4)
expr : expr PLUS expr . (4)

PLUS  shift 10
RPAR  reduce 4
TERM  reduce 4
```

On retrouve qu'effectivement la situation $E \pm E \bullet \pm E$ amène à se poser la question si la production $E \rightarrow E \pm E$ doit être appliquée sur le \pm de gauche (cas réduction) ou le \pm de droite (cas shift).

Question 3. En remettant la précéence `%left PLUS`, on peut constater par notre affichage que le \pm de gauche est préféré (donc la réduction).

```
$ ./tp
1 + 2 + 3 ;;
((1) + (2)) + (3)
```

Question 4. En modifiant la précéence en `%right PLUS`, on peut constater par notre affichage que le \pm de droite est préféré (donc le shift).

```
$ ./tp
1 + 2 + 3 ;;
(1) + ((2) + (3))
```

Question 5. Laisser finalement `%left PLUS`; nous reviendrons sur les précéences plus loin dans le TP.

— La déclaration de l'axiome :

```
%type <Ast.expr> ansyn
%start ansyn
```

où `ansyn` correspond au non-terminal S de la grammaire. La première ligne permet de préciser qu'à une dérivation de S , on souhaite associer un AST tel que nous l'avons défini dans `ast.ml`.

— La spécification de la grammaire.

```
ansyn:
  | TERM ansyn          { $2 }
  | expr TERM           { $1 }
  ;

expr:
  | INT                 { ECst $1 }
  | expr PLUS expr      { EAdd ($1, $3) }
  | LPAR expr RPAR       { $2 }
  ;
```

Ici `expr` correspond au non-terminal E . On retrouve entre accolades la correspondance entre les productions et le constructeur de l'AST associé.

1.4 Analyseur lexical

Le fichier `lexer.mll` contient la spécification de l'analyseur lexical qui nous intéresse. Le fichier suit l'organisation présentée à l'adresse <https://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html#sec298>.

On retrouve dans ce fichier la spécification des unités lexicales côté lexème. Dans le cas de notre grammaire, voici le dictionnaire que l'on souhaite implémenter.

Token		Lexème	Attribut	
grammaire	parser.mly	expression régulière	type	conversion
<u>int</u>	INT	[0-9]+	int	int_of_string
<u>+</u>	PLUS	"+"		
<u>(</u>	LPAR	"("		
<u>)</u>	RPAR	")"		
<u>;;</u>	TERM	";;"		

On retrouve ces correspondances dans la spécification de l'analyseur lexical.

```
rule anlex = parse
| [' ' '\t' '\n' '\r']      { anlex lexbuf (* Oubli des... *) }
| ['0'-'9']+ as lxm        { INT(int_of_string lxm) }
| '+'                      { PLUS }
| '('                      { LPAR }
| ')'                      { RPAR }
| ";;"                     { TERM }
| eof                      { raise Eof }
| _ as lxm                  { (* Pour tout autre caractère : ... *)
                             Printf.eprintf
                               "Unknown character '%c': ignored\n"
                               lxm;
                             flush stderr;
                             anlex lexbuf
                           }
```

On remarquera 3 lignes supplémentaires qui ne sont pas présentes dans le tableau précédent.

1. `[' ' '\t' '\n' '\r']` : Il s'agit de la spécification des caractères d'espace ; lorsque l'un d'eux est rencontré, l'analyseur est rappelé (`anlex lexbuf`) ; ils sont ainsi reconnus mais non transmis à l'analyseur syntaxique (aucun token ne leur est associé).
2. `eof` : Il s'agit du caractère spécial de fin de fichier indiquant que plus aucun caractère n'est attendu ; on lève une exception qui sera utilisée dans le programme principal.
3. `_ as lxm` : Pour tout autre caractère, celui-ci est considéré inconnu et sera ignoré ; un *warning* est affiché et l'analyseur est rappelé.

2 Ajout de la multiplication

Les fichiers du projet sont fonctionnels mais limités à une grammaire simple que nous allons étendre, en commençant par l'ajout de la multiplication. L'ajout d'une construction passe par l'extension des trois fichiers principaux. L'idéal est de procéder de l'abstrait vers le concret, c'est-à-dire :

1. `ast.ml` : ajouter la construction à la syntaxe abstraite
2. `parser.mly` : étendre la grammaire (déclarations des tokens nécessaires et des productions)
3. `lexer.mll` : ajouter la spécification des unités lexicales manquantes

Dans notre cas, on cherche à étendre la grammaire avec la production suivante.

$$E \rightarrow E * E$$

2.1 Extension de la syntaxe abstraite

Dans le fichier `ast.ml`.

Question 1. Étendre le type `expr` avec une nouvelle construction `EMul` pour la multiplication ; s'inspirer de l'addition.

Question 2. Étendre la fonction `string_of_expr` pour gérer la nouvelle construction.

2.2 Extension de la grammaire

Dans le fichier `parser.mly` :

Question 3. Ajouter la définition d'un token `TIME` pour le symbole `*`.

Question 4. Ajouter la production $E \rightarrow E * E$ à l'image de l'addition ; veillez à associer le constructeur `EMul` à la production.

A ce stade, la compilation doit produire 3 conflits shift/reduce.

```
$ make
ocamlyacc -v parser.mly
3 shift/reduce conflicts.
ocamldep ast.ml parser.ml lexer.ml main.ml > depend
ocamlc -o parser.cmo -c parser.ml
ocamlc -o lexer.cmo -c lexer.ml
ocamlc -o main.cmo -c main.ml
ocamlc -o tp ast.cmo parser.cmo lexer.cmo main.cmo
```

Dans le détail (fichier `parser.output`) :

```
14: shift/reduce conflict (shift 11, reduce 4) on TIME
state 14
  expr : expr . PLUS expr  (4)
  expr : expr PLUS expr .  (4)
  expr : expr . TIME expr  (5)
```

```

TIME  shift 11
PLUS  reduce 4
RPAR  reduce 4
TERM  reduce 4

```

```

15: shift/reduce conflict (shift 10, reduce 5) on PLUS
15: shift/reduce conflict (shift 11, reduce 5) on TIME
state 15

```

```

    expr : expr . PLUS expr (4)
    expr : expr . TIME expr (5)
    expr : expr TIME expr . (5)

```

```

PLUS  shift 10
TIME  shift 11
RPAR  reduce 5
TERM  reduce 5

```

Les trois conflits sont les suivants.

- shift sur $E \bullet \underline{*} E$ vs. reduce de $E \underline{+} E \bullet$: la situation est donc $E \underline{+} E \bullet \underline{*} E$
- shift sur $E \bullet \underline{+} E$ vs. reduce de $E \underline{*} E \bullet$: la situation est donc $E \underline{*} E \bullet \underline{+} E$
- shift sur $E \bullet \underline{*} E$ vs. reduce de $E \underline{*} E \bullet$: la situation est donc $E \underline{*} E \bullet \underline{*} E$

Les deux premiers conflits indiquent que la grammaire ne précise pas qui est prioritaire entre la multiplication et l'addition. Le dernier conflit est similaire à celui traité précédemment pour l'addition.

Tout comme pour l'addition, nous allons préciser que nous préférons une associativité gauche en ajoutant `%left TIME` comme précédence pour la multiplication. De plus, nous allons ajouter que la priorité de la multiplication sur l'addition en ajoutant cette précédence après celle de `PLUS`. Cela donne au final dans le fichier `parser.mly`.

```

...
%left PLUS
%left TIME
...

```

Après compilation, les conflits sont résolus.

2.3 Extension du lexique

Le fichier `lexer.mll` est simplement étendu avec une nouvelle ligne.

```
| '*' { TIME }
```

3 Ajout de la soustraction (sucre syntaxique)

L'objectif est de rajouter à la syntaxe concrète la production suivante pour la pouvoir écrire des soustractions.

$$E \rightarrow E \underline{-} E$$

Cependant, plutôt que de rajouter la soustraction directement dans la syntaxe abstraite, nous allons utiliser l'addition déjà présente à travers la propriété

$$a - b = a + (-b)$$

qui remplace une soustraction par une addition avec la négation de l'opérande droite.

Question 1. Ajouter la construction `ENeg` dans `ast.ml` pour la négation (pas la soustraction, mais l'opération *unaire*) que nous n'avons pas.

Question 2. Ajouter le token `MINUS` et la production $E \rightarrow E _ E$ dans `parser.mly` où la soustraction est traduite à travers la propriété $a - b = a + (-b)$.

Question 3. Ajouter la règle pour reconnaître le caractère `-` dans `lexer.mll`.

Vous devriez rencontrer les conflits correspondants aux situations :

- $E _ E \bullet _ E$
- $E _ E \bullet + E$
- $E + E \bullet _ E$
- $E _ E \bullet * E$
- $E * E \bullet _ E$

Vous devez alors préciser que la négation est moins prioritaire que la multiplication. Elle est considérée prioritaire sur la gauche (`%left`) et de priorité équivalente que l'addition (donc sur la même ligne dans le fichier).

4 Ajout de la négation

Question 1. Procéder de la même façon que précédemment pour ajouter la négation unaire, c'est-à-dire la production

$$E \rightarrow _ E$$

qu'il suffit simplement de spécifier dans `parser.mly` en l'associant à la construction `ENeg` déjà définie dans `ast.ml`.

Bien qu'aucune collision soit détectée par `ocamlyacc`, un problème apparaît à l'exécution.

```
$ ./tp
- 1 * 2 ;;
- ((1) * (2))
```

alors que l'on s'attend à obtenir $(- (1)) * (2)$. En effet, la négation est plus prioritaire que l'addition, la soustraction et la multiplication.

Le problème vient que l'opérateur de négation est le même que celui de la soustraction (`MINUS`) ; la négation hérite de la précedence spécifiée pour la soustraction et aucune collision n'apparaît à la compilation. Pour remédier à ce problème, nous créons une nouvelle précedence `NEG` spécifiquement pour la négation.

```
%left PLUS MINUS
%left TIME
%nonassoc NEG
```

```
...
| MINUS expr      %prec NEG  { ENeg $2 }
```

La précedence NEG est définie à un niveau plus prioritaire que TIME, PLUS et MINUS comme attendu. Comme la négation n'est pas un opérateur binaire, les annotations %left et %right n'ont pas de sens ; on utilise %nonassoc à la place. Enfin l'annotation %prec NEG utilisée dans la production indique qu'il ne faut plus utiliser la précedence de MINUS mais bien celle de NEG.

Question 2. Faire en sorte que votre analyseur gère correctement les précédences comme il vient d'être décrit.

5 Ajout des identifiants

Question. Ajouter la grammaire la production

$$E \rightarrow \underline{\text{id}}$$

où id correspond à des identifiants (suite de caractères commençant par une lettre puis composés de lettres, de chiffres et d'*underscores*).

6 Ajout des commentaires

Il est possible de définir dans `lexer.mll` plusieurs jeux de règles s'appelant les uns les autres. On peut utiliser cette technique pour programmer les commentaires. Pour cela, il suffit de lancer un jeu de règles propres aux commentaires, de l'appeler lors de la reconnaissance de l'ouverture d'un commentaire, puis de retourner le jeu de règles principal. Cela s'implémente de la façon suivante.

```
rule anlex = parse
| [' ' '\t' '\n' '\r']      { anlex lexbuf (* ... *) }
| "/*"                      { comment lexbuf }
...
| eof                       { raise Eof }
| _ as lxm                  { (* Pour tout autre caractère : ... *)
                             Printf.eprintf
                               "Unknown character '%c': ignored\n"
                               lxm;
                             flush stderr;
                             anlex lexbuf
                           }

and comment = parse
| "*/"                      { anlex lexbuf }
| _                          { comment lexbuf }
```

Question. Intégrer les commentaires à votre analyseur.

7 Ajout d'autres types de constantes

Dans `ast.ml`, créer le type `value` pour manipuler d'autres types de constantes.

```
type value =
  | VInt of int
  | ...

type expr =
  | ECst of value
  | ...
```

Question 1. Ajouter les booléens, les valeurs flottantes, ainsi que les chaînes de caractères au type `value`.

Question 2. Définir la fonction `string_of_value` et mettre à jour `string_of_expr`.

Question 3. Mettre à jour la règle de production $E \rightarrow \underline{\text{int}}$ dans `ast.ml` de la façon suivante.

$$\begin{array}{ll}
 E \rightarrow V & \text{(constante)} \\
 | \dots & \\
 \\
 V \rightarrow \underline{\text{int}} & \text{(constante entière)} \\
 | \underline{\text{true}} & \text{(constante booléenne vrai)} \\
 | \underline{\text{false}} & \text{(constante booléenne faux)} \\
 | \underline{\text{float}} & \text{(constante flottante)} \\
 | \underline{\text{str}} & \text{(chaîne de caractères)}
 \end{array}$$

Question 4. Ajouter les spécifications des unités lexicales correspondantes dans le fichier `lexer.mll`.