

projet de compilation  
**Mini Modules**

version 1c — 19 novembre 2009

L'objectif de ce projet est de réaliser un compilateur pour un fragment de Caml, appelé Mini Modules par la suite, produisant du code MIPS. Il s'agit d'un fragment contenant des entiers, des booléens, des listes, des références et un système de modules et de foncteurs. En revanche, il n'y a ni inférence de types, ni ordre supérieur (de fonctions ou de modules). Il s'agit d'un fragment 100% compatible avec Caml, au sens où tout programme de Mini Modules est aussi un programme Caml correct. Ceci permettra notamment d'utiliser ce dernier comme référence. Le présent sujet décrit précisément Mini Modules, ainsi que la nature du travail demandé.

## 1 Analyse lexicale et syntaxique

Dans la suite, nous utilisons les notations suivantes dans les grammaires :

$\langle \text{r\`egle} \rangle^*$	répétition de la règle $\langle \text{r\`egle} \rangle$ un nombre quelconque de fois (y compris aucune)
$\langle \text{r\`egle} \rangle_t^*$	répétition de la règle $\langle \text{r\`egle} \rangle$ un nombre quelconque de fois (y compris aucune), les occurrences étant séparées par le terminal $t$
$\langle \text{r\`egle} \rangle^+$	répétition de la règle $\langle \text{r\`egle} \rangle$ au moins une fois
$\langle \text{r\`egle} \rangle_t^+$	répétition de la règle $\langle \text{r\`egle} \rangle$ au moins une fois, les occurrences étant séparées par le terminal $t$
$\langle \text{r\`egle} \rangle?$	utilisation optionnelle de la règle $\langle \text{r\`egle} \rangle$ (i.e. 0 ou 1 fois)
$( \langle \text{r\`egle} \rangle )$	parenthésage; attention à ne pas confondre ces parenthèses avec les terminaux ( et )

### 1.1 Analyse lexicale

Espaces, tabulations et retour-chariots constituent les blancs. Les commentaires débutent par  $(*$  et s'étendent jusqu'à  $*)$ , et ils peuvent être imbriqués. On distingue deux catégories d'identificateurs, définies par les expressions régulières  $\langle \text{lident} \rangle$  et  $\langle \text{uident} \rangle$  suivantes :

$$\begin{aligned} \langle \text{chiffre} \rangle & ::= 0-9 \\ \langle \text{alpha} \rangle & ::= \text{a-z} \mid \text{A-Z} \\ \langle \text{lident} \rangle & ::= (\text{a-z}) (\langle \text{alpha} \rangle \mid \_ \mid ' \mid \langle \text{chiffre} \rangle)^* \\ \langle \text{uident} \rangle & ::= (\text{A-Z}) (\langle \text{alpha} \rangle \mid \_ \mid ' \mid \langle \text{chiffre} \rangle)^* \end{aligned}$$

Les identificateurs suivants sont des mots clés :

```

else  end    false  if      in
let   match  mod     module  open
rec   sig    struct  then    true
val   with

```

Les constantes entières obéissent à l'expression régulière  $\langle entier \rangle$  suivante :

$$\langle entier \rangle ::= \langle chiffre \rangle^+$$

## 1.2 Analyse syntaxique

La grammaire des fichiers sources considérée est donnée figure 1. Le point d'entrée est le non-terminal  $\langle fichier \rangle$ .

```

<fichier>      ::= <elt>* EOF
<elt>          ::= let () = <expr>
                | let rec? <lident> <farg>* : <type> = <expr>
                | module <uident> <marg>* (: <signature>)? = <mexpr>
                | open <upath>
<type>         ::= unit | bool | int | <type> ref | <type> list
<farg>         ::= ( <lident> : <type> )
<signature>    ::= sig <decl>* end
<decl>         ::= val <lident> : <type>+
                | module <uident> <marg>* : <signature>
<marg>         ::= ( <uident> : <signature> )
<mexpr>        ::= struct <elt>* end
                | <upath> (( <upath> ))*

<simple_expr>   ::= ( <expr> ) | <lpath> | <const> | ! <simple_expr>
<expr>         ::= <simple_expr>
                | <lpath> <simple_expr>+
                | - <expr> | <expr> <binop> <expr>
                | if <expr> then <expr> else <expr>
                | if <expr> then <expr>
                | let <lident> = <expr> in <expr>
                | match <expr> with |? [] -> <expr> | <lident> :: <lident> -> <expr>
<binop>        ::= + | - | * | / | mod | <= | >= | > | < | <> | =
                | && | || | :: | := | ;
<const>        ::= true | false | <entier> | () | []
<upath>        ::= <uident>+
<lpath>        ::= (<upath> .)? <lident>

```

FIG. 1 – Grammaire des fichiers de Mini Modules

Les associativités et précédences des diverses constructions sont données par la table suivante, de la plus faible à la plus forte précedence :

opérateur ou construction	associativité
let in	—
match with	—
;	à gauche
if then else	—
:=	à droite
	à gauche
&&	à gauche
< <= > >= = <>	à gauche
::	à droite
+ -	à gauche
* / mod	à gauche
- (unaire)	—
application de fonction	—

La construction `if  $e_1$  then  $e_2$`  est du sucre syntaxique pour `if  $e_1$  then  $e_2$  else ()`, comme en Caml.

## 2 Analyse sémantique

### 2.1 Types et environnements de typage

Dans tout ce qui suit, les expressions de types sont de la forme suivante :

$$\tau ::= \text{unit} \mid \text{bool} \mid \text{int} \mid \tau \text{ ref} \mid \tau \text{ list}$$

Un environnement de typage  $\Gamma$  est une suite ordonnée de déclarations  $x : \delta$ , avec  $x$  un identificateur et  $\delta$  de la forme suivante :

$$\delta ::= \text{val } \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau \quad \text{avec } n \geq 0 \\ \mid \text{module } \Gamma_1 \rightarrow \dots \rightarrow \Gamma_n \rightarrow \Gamma \quad \text{avec } n \geq 0$$

Un nom qualifié  $p$  est de la forme  $x_1 \dots x_n$ , où les  $x_i$  sont des identificateurs et  $n \geq 1$ . On note  $\Gamma(p)$  l'accès à  $p$  dans l'environnement  $\Gamma$ , ainsi défini:

$$\Gamma(x) = \delta \quad \text{si } x : \delta \in \Gamma \\ \Gamma(p.x) = \Gamma'(x) \quad \text{si } \Gamma(p) = \text{module } \Gamma'$$

**Primitives.** On supposera par la suite que les programmes sont toujours typés dans l'environnement initial suivant :

```
print_int : int → unit
print_newline : unit → unit
read_int : unit → int
not : bool → bool
```

Une cinquième primitive est `ref`, dont le typage est particulier car elle est polymorphe (voir les règles de typage données dans la section suivante). On notera que ces cinq primitives ne sont pas des mots clés. Par conséquent, elles peuvent être (localement) masquées par des définitions du programme.

## 2.2 Typage des expressions

On introduit le jugement  $\Gamma \vdash e : \tau$  signifiant « dans l'environnement  $\Gamma$ , l'expression  $e$  est bien typée de type  $\tau$  » et on le définit par les règles d'inférence suivantes :

$$\begin{array}{c}
\frac{c \text{ constante de type } \tau}{\Gamma \vdash c : \tau} \quad \frac{\Gamma(p) = \mathbf{val} \ \tau}{\Gamma \vdash p : \tau} \quad \frac{\Gamma \vdash e : \mathbf{int}}{\Gamma \vdash - e : \mathbf{int}} \\
\\
\frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int} \quad op \in \{=, <>, <=, >=, <, >\}}{\Gamma \vdash e_1 \ op \ e_2 : \mathbf{bool}} \\
\\
\frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int} \quad op \in \{+, -, *, /\}}{\Gamma \vdash e_1 \ op \ e_2 : \mathbf{int}} \\
\\
\frac{\Gamma \vdash e_1 : \mathbf{bool} \quad \Gamma \vdash e_2 : \mathbf{bool} \quad op \in \{\&\&, ||\}}{\Gamma \vdash e_1 \ op \ e_2 : \mathbf{bool}} \\
\\
\frac{\Gamma \vdash e_1 : \mathbf{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathbf{if} \ e_1 \ \mathbf{then} \ e_2 \ \mathbf{else} \ e_3 : \tau} \\
\\
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let} \ x=e_1 \ \mathbf{in} \ e_2 : \tau_2} \\
\\
\frac{}{\Gamma \vdash [] : \tau \ \mathbf{list}} \quad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \ \mathbf{list}}{\Gamma \vdash e_1 :: e_2 : \tau \ \mathbf{list}} \\
\\
\frac{\Gamma \vdash e_1 : \tau' \ \mathbf{list} \quad \Gamma \vdash e_2 : \tau \quad \Gamma, x_1 : \tau', x_2 : \tau' \ \mathbf{list} \vdash e_3 : \tau \quad x_1 \neq x_2}{\Gamma \vdash \mathbf{match} \ e_1 \ \mathbf{with} \ [] \rightarrow e_2 \mid x_1 :: x_2 \rightarrow e_3 : \tau} \\
\\
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{ref} \ e : \tau \ \mathbf{ref}} \quad \frac{\Gamma \vdash e : \tau \ \mathbf{ref}}{\Gamma \vdash !e : \tau} \quad \frac{\Gamma \vdash e_1 : \tau \ \mathbf{ref} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \mathbf{unit}} \\
\\
\frac{\Gamma(p) = \mathbf{val} \ \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau \quad \Gamma \vdash e_i : \tau_i}{\Gamma \vdash p \ e_1 \ \dots \ e_n : \tau}
\end{array}$$

On notera qu'à la différence de Caml, la construction `let in` ne généralise pas le type de son premier argument. En particulier, l'expression

```
let l = [] in let x = 1 :: l in let y = true :: l in ()
```

n'est pas bien typée dans Mini Modules.

## 2.3 Typage des modules

Une définition d'un élément de module est notée  $d$  et peut prendre quatre formes :

$$\begin{array}{l}
d ::= \mathbf{let} \ () = e \\
\quad | \ \mathbf{let} \ x \ (x_1 : \tau_1) \dots (x_n : \tau_n) : \tau = e \\
\quad | \ \mathbf{let} \ \mathbf{rec} \ x \ (x_1 : \tau_1) \dots (x_n : \tau_n) : \tau = e \\
\quad | \ \mathbf{module} \ x \ (x_1 : S_1) \dots (x_n : S_n) = m \\
\quad | \ \mathbf{module} \ x \ (x_1 : S_1) \dots (x_n : S_n) : S = m
\end{array}$$

Une signature  $S$  est une liste d'éléments de signature. Un élément de signature est noté  $s$  et peut prendre deux formes :

$$s ::= \text{val } x : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau \\ | \text{module } x(x_1 : S_1) \dots (x_n : S_n) : S$$

On note  $\bar{s}$  la déclaration d'environnement correspondant à  $s$ , définie par

$$\overline{\text{val } x : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau} = x : \text{val } \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau \\ \overline{\text{module } x(x_1 : S_1) \dots (x_n : S_n) : S} = x : \text{module } \bar{S}_1 \rightarrow \dots \rightarrow \bar{S}_n \rightarrow \bar{S} \\ \overline{\text{sig } s_1 \dots s_n \text{ end}} = \bar{s}_1, \dots, \bar{s}_n$$

Enfin, pour une définition  $d$ , on note  $\bar{d}$  la déclaration d'environnement correspondante, le cas échéant (la définition  $\text{let } () = e$  ne correspond pas à une déclaration).

**Définitions.** On introduit le jugement  $\Gamma_1 \vdash d \Rightarrow \Gamma_2$  pour le typage des définitions, signifiant « dans l'environnement  $\Gamma_1$ , la définition est bien formée et étend  $\Gamma_1$  en un nouvel environnement  $\Gamma_2$  ». On le définit par les règles d'inférence suivantes :

$$\frac{\Gamma \vdash e : \text{unit}}{\Gamma \vdash \text{let } () = e \Rightarrow \Gamma} \\ \frac{\Gamma, x_1 : \tau_1, \dots, x_n : \tau_n \vdash e : \tau}{\Gamma \vdash \text{let } x(x_1 : \tau_1) \dots (x_n : \tau_n) : \tau = e \Rightarrow \Gamma, x : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau} \\ \frac{n \geq 1 \quad \Gamma, x : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau, x_1 : \tau_1, \dots, x_n : \tau_n \vdash e : \tau}{\Gamma \vdash \text{let rec } x(x_1 : \tau_1) \dots (x_n : \tau_n) : \tau = e \Rightarrow \Gamma, x : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau} \\ \frac{\Gamma, x_1 : \bar{S}_1, \dots, x_n : \bar{S}_n \vdash m : \Gamma'}{\Gamma \vdash \text{module } x(x_1 : S_1) \dots (x_n : S_n) = m \Rightarrow \Gamma, x : \Gamma_1 \rightarrow \dots \rightarrow \Gamma_n \rightarrow \Gamma'} \\ \frac{\Gamma, x_1 : \bar{S}_1, \dots, x_n : \bar{S}_n \vdash m : \Gamma' \quad \Gamma' \sqsubseteq \bar{S}}{\Gamma \vdash \text{module } x(x_1 : S_1) \dots (x_n : S_n) : S = m \Rightarrow \bar{S}}$$

**Expressions de modules.** Une expression de module  $m$  est définie par

$$m ::= \text{struct } d_1 \dots d_n \text{ end} \\ | x(x_1) \dots (x_n)$$

On introduit le jugement  $\Gamma \vdash m : \Gamma'$  signifiant «  $m$  est bien formée dans l'environnement  $\Gamma$  et de signature  $\Gamma'$  ». On le définit par les règles suivantes :

$$\overline{\Gamma \vdash \text{struct end} : \emptyset} \\ \frac{\Gamma \vdash d_1 \Rightarrow \Gamma_1 \quad \Gamma_1 \vdash \text{struct } d_2 \dots d_n \text{ end} : \Gamma'}{\Gamma \vdash \text{struct } d_1 \dots d_n \text{ end} : \bar{d}_1, \Gamma'} \\ \frac{\Gamma(p) = \text{module } \Gamma_1 \rightarrow \dots \rightarrow \Gamma_n \rightarrow \Gamma' \quad \Gamma(p_i) = \text{module } \Gamma'_i \quad \Gamma'_i \sqsubseteq \Gamma_i}{\Gamma \vdash p(p_1) \dots (p_n) : \Gamma'}$$

**Sous-typage des signatures.** Pour deux signatures  $\Gamma_1$  et  $\Gamma_2$ , on note  $\Gamma_1 \sqsubseteq \Gamma_2$  la relation signifiant que « tout module réalisant la signature  $\Gamma_1$  est acceptable là où un module de signature  $\Gamma_2$  est attendu ». Cette relation est ainsi définie :

$$\frac{\Gamma(x_i) = \delta_i \quad \delta_i \sqsubseteq \delta'_i}{\Gamma \sqsubseteq x_1 : \delta'_1, \dots, x_n : \delta'_n}$$

où le sous-typage entre déclarations est défini par

$$\frac{\overline{\text{val } \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau \sqsubseteq \text{val } \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau}}{\frac{\Gamma'_i \sqsubseteq \Gamma_i \quad \Gamma \sqsubseteq \Gamma'}{\text{module } \Gamma_1 \rightarrow \dots \rightarrow \Gamma_n \rightarrow \Gamma \sqsubseteq \text{module } \Gamma'_1 \rightarrow \dots \rightarrow \Gamma'_n \rightarrow \Gamma'}}$$

**Règles d'unicité.** On ajoute aux règles précédentes la condition d'unicité suivante : une signature ou une structure ne peut contenir deux modules ou deux valeurs de même nom.

**Typage des fichiers.** Un fichier est un ensemble de définitions, qui sont typées exactement comme le contenu d'un module encadré par `struct end`.

### 3 Compilation des modules

On se propose ici de compiler les modules par *défunctorisation*. Cette opération consiste à remplacer toute application de foncteur  $F(M_1) \dots (M_n)$  par le module obtenu en substituant dans le corps du foncteur  $F$  les arguments formels de  $F$  par les arguments effectifs  $M_1, \dots, M_n$ . Au final, il ne reste que des modules, mais plus de foncteurs. Les modules peuvent alors être ignorés dans le processus de compilation, car ils ne participent qu'à la structuration de l'espace de noms. Au prix du renommage de certains identificateurs, on peut même supprimer les modules eux-mêmes, pour ne conserver qu'une suite linéaire de déclarations de variables et de fonctions.

Ainsi, on peut supprimer les foncteurs dans le programme suivant

```
module Double(X : sig val f : int -> int end) = struct
  let f (x:int) : int = X.f (X.f x)
end
module X2 = struct let f (x:int) : int = x*x end
module X4 = Double(X2)
module X16 = Double(X4)
```

pour obtenir le programme suivant :

```
module X2 = struct let f (x:int) : int = x*x end
module X4 = struct let f (x:int) : int = X2.f (X2.f x) end
module X16 = struct let f (x:int) : int = X4.f (X4.f x) end
```

En renommant les trois fonctions `f` on obtient enfin le programme suivant :

```
let f1 (x:int) : int = x*x
let f2 (x:int) : int = f1 (f1 x)
let f3 (x:int) : int = f2 (f2 x)
```

La phase de défonctorisation pourra être insérée entre l'analyse sémantique et la production de code.

## 4 Production de code

L'objectif est de réaliser un compilateur simple mais correct. En particulier, on ne cherchera pas à faire d'allocation de registres mais on se contentera d'utiliser la pile pour stocker les éventuels calculs intermédiaires. Bien entendu, il est possible, et même souhaitable, d'utiliser localement les registres de MIPS.

La sémantique est celle de Caml. En particulier, l'ordre d'évaluation des arguments d'une fonction ou des éléments d'une liste n'est pas spécifié. On notera que les règles de typage limitent l'utilisation des opérateurs de comparaison aux entiers. Il n'est donc pas demandé de réaliser une égalité structurelle comme en Caml. On ne cherchera pas à libérer la mémoire allouée pour les listes.

Le schéma de compilation pourra être le suivant. Toutes les valeurs contenues dans des variables sont stockées dans un mot de 32 bits ; plus précisément :

- les entiers sont directement représentés par des entiers 32 bits (toute petite différence avec Caml) ;
- les constantes `()`, `false` et `[]` sont représentées par l'entier 0 et la constante `true` par l'entier 1 ;
- une référence est représentée par un pointeur vers une donnée allouée sur le tas contenant un mot de 32 bits ;
- une liste autre que `[]` est représenté par un pointeur vers une donnée allouée sur le tas, contenant deux mots de 32 bits.

En particulier, le filtrage sur les listes est réalisé en comparant la valeur filtrée avec 0, qui représente nécessairement `[]` (toute valeur allouée sur le tas sera un pointeur différent de 0).

**Remarque importante.** La correction du projet sera réalisée en partie automatiquement, à l'aide d'un jeu de petits programmes réalisant des affichages (avec `print_int` et `print_newline`), qui seront compilés avec votre compilateur et dont la sortie sera comparée à la sortie attendue. Il est donc très important :

- de correctement compiler les appels à `print_int` et `print_newline` ;
- de respecter la sémantique de ces fonctions (pas de retour-chariot après `print_int` notamment).

## 5 Travail demandé

Écrire un compilateur, appelons-le `mimo`, acceptant sur sa ligne de commande exactement un fichier Mini Modules (portant l'extension `.ml`) et éventuellement l'option

`-parse-only` ou l'option `-type-only`. Ces deux options indiquent respectivement de stopper la compilation après l'analyse syntaxique et l'analyse sémantique.

Si le fichier est conforme à la syntaxe et au typage décrits dans ce document, votre compilateur doit produire du code MIPS dans un fichier (portant le même nom que le fichier source mais avec le suffixe `.s`) et terminer avec le code de sortie 0 (`exit 0` explicite ou terminaison normale du programme), sans rien afficher.

En cas d'erreur lexicale, syntaxique ou de typage, celle-ci doit être signalée (de la manière indiquée ci-dessous) et le programme doit terminer avec le code de sortie 1 (`exit 1`). En cas d'autre erreur (une erreur du compilateur lui-même), le programme doit terminer avec le code de sortie 2 (`exit 2`).

**Localisation des erreurs.** Lorsqu'une erreur est détectée par votre compilateur, elle doit être signalée le plus précisément possible, par sa nature et sa localisation dans le fichier source. On adoptera le format suivant pour cette signalisation :

```
File "test.ml", line 4, characters 5-6:  
syntax error
```

L'anglicisme de la première ligne est nécessaire pour que la fonction `next-error` d'Emacs puisse interpréter la localisation et placer ainsi automatiquement le curseur sur l'emplacement de l'erreur. (En revanche, le message d'erreur proprement dit pourra être écrit en français si vous le souhaitez.)

Les localisations peuvent être obtenues pendant l'analyse syntaxique grâce aux mots-clés `$startpos` et `$endpos` de Menhir, puis conservées dans l'arbre de syntaxe abstraite.

**Modalités de remise de votre projet.** Votre projet doit se présenter sous forme d'une archive tar compressée (option "z" de tar), appelée `vos_noms.tgz` qui doit contenir un répertoire appelé `vos_noms` (exemple : `dupont-durand.tgz`). Dans ce répertoire doivent se trouver les *sources* de votre programme (inutile d'inclure les fichiers compilés). Quand on se place dans ce répertoire, la commande `make` doit créer votre compilateur, qui sera appelé `mimo`. La commande `make clean` doit effacer tous les fichiers que `make` a engendrés et ne laisser dans le répertoire que les fichiers sources.

L'archive doit également contenir un court rapport expliquant les différents choix techniques qui ont été faits et, le cas échéant, les difficultés rencontrées ou les éléments non réalisés. Ce rapport pourra être fourni dans un format ASCII, PostScript ou PDF.

Le projet est à faire seul ou en binôme. Il doit être remis par email à `filliatr@lri.fr`. La date limite de remise sera annoncée sur le site du cours.