projet de compilation

Petit Caml

version 3

L'objectif de ce projet est de réaliser un compilateur pour un fragment de Caml, appelé Petit Caml par la suite, produisant du code MIPS. Il s'agit d'un fragment contenant des entiers, des booléens, des listes, des n-uplets, de l'ordre supérieur, du filtrage et de l'inférence de types. Il s'agit d'un fragment 100% compatible avec Objective Caml, ce qui permettra d'utiliser ce dernier comme référence. Le présent sujet décrit précisément Petit Caml, ainsi que la nature du travail demandé.

1 Analyse lexicale et syntaxique

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

$\langle règle \rangle^\star$	répétition de la règle $\langle règle \rangle$ un nombre quelconque de fois (y compris aucune)
$\langle règle \rangle_t^\star$	répétition de la règle $\langle règle \rangle$ un nombre quelconque de fois (y compris aucune), les occurrences étant séparées par le terminal t
$\langle règle \rangle^+$	répétition de la règle $\langle règle \rangle$ au moins une fois
$\langle règle \rangle_t^+$	répétition de la règle $\langle règle \rangle$ au moins une fois, les occurrences étant séparées par le terminal t
$\langle règle \rangle$?	utilisation optionnelle de la règle $\langle règle \rangle$ (i.e. 0 ou 1 fois)
($\langle règle \rangle$)	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. Les identificateurs obéissent à l'expression régulière $\langle ident \rangle$ suivante :

$$\begin{array}{lll} \langle chiffre \rangle & ::= & \texttt{0–9} \\ \langle alpha \rangle & ::= & \texttt{a–z} \mid \texttt{A–Z} \\ \langle ident \rangle & ::= & \texttt{(a–z)}\ (\langle alpha \rangle \mid \texttt{\_} \mid \texttt{'} \mid \langle chiffre \rangle)^\star \end{array}$$

Les identificateurs suivants sont des mots clés :

```
else     false    function    if
in       let      match       not
rec      then     true        with
```

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

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

Les chaînes de caractères sont délimitées par le caractère `"`. Elles peuvent contenir n'importe quels caractères, à l'exception de `"`, `\` et du retour-chariot. Ces trois caractères doivent être encodés (à l'intérieur d'une chaîne) par les séquences `\"`, `\\` et `\n`, respectivement.

1.2 Analyse syntaxique

La grammaire des fichiers sources considérée est donnée figure 1. Le point d'entrée est le non-terminal ⟨*fichier*⟩.

⟨*fichier*⟩	::=	⟨*decl*⟩* `EOF`
⟨*decl*⟩	::=	`let` ⟨*motif*⟩ `=` ⟨*expr*⟩
	|	`let rec?` ⟨*ident*⟩ ⟨*motif*⟩$^+$ `=` ⟨*expr*⟩
⟨*motif*⟩	::=	`_` | ⟨*ident*⟩ | `(` ⟨*motif*⟩ `,` ⟨*motif*⟩$^+_,$ `)`
⟨*simple_expr*⟩	::=	`(` ⟨*expr*⟩ `)` | ⟨*ident*⟩ | ⟨*const*⟩
	|	`(` ⟨*expr*⟩ `,` ⟨*expr*⟩$^+_,$ `)` | `[` ⟨*expr*⟩$^\star_;$ `]`
⟨*expr*⟩	::=	⟨*simple_expr*⟩
	|	⟨*simple_expr*⟩ ⟨*simple_expr*⟩$^+$
	|	`function` ⟨*motif*⟩ `->` ⟨*expr*⟩
	|	⟨*unop*⟩ ⟨*expr*⟩ | ⟨*expr*⟩ ⟨*op*⟩ ⟨*expr*⟩ | ⟨*expr*⟩ `::` ⟨*expr*⟩
	|	`if` ⟨*expr*⟩ `then` ⟨*expr*⟩ `else` ⟨*expr*⟩
	|	`let` ⟨*motif*⟩ `=` ⟨*expr*⟩ `in` ⟨*expr*⟩
	|	`let rec?` ⟨*ident*⟩ ⟨*motif*⟩$^+$ `=` ⟨*expr*⟩ `in` ⟨*expr*⟩
	|	`match` ⟨*expr*⟩ `with []` `->` ⟨*expr*⟩ `|` ⟨*motif*⟩ `::` ⟨*motif*⟩ `->` ⟨*expr*⟩
⟨*op*⟩	::=	`+` | `-` | `*` | `/`
	|	`<=` | `>=` | `>` | `<` | `<>` | `=` | `&&` | `||`
⟨*unop*⟩	::=	`-` | `not`
⟨*const*⟩	::=	`true` | `false` | ⟨*entier*⟩ | ⟨*chaîne*⟩ | `()`

FIG. 1 – Grammaire des fichiers de Petit Caml

Les associativités et précédences des divers opérateurs ou construction sont données par la table suivante, de la plus faible à la plus forte précédence :

opérateur ou construction	associativité
`in`	—
`else`	—
`->`	—
`||`	à gauche
`&&`	à gauche
`<  <=  >  >=  =  <>`	à gauche
`::`	à droite
`+  -`	à gauche
`*  /`	à gauche
`-` (unaire)	—
`not`	—
application de fonction	à gauche

La notation `let` $f\ p_1 \dots p_n = e$ est, comme un Caml, un raccourci pour `let` $f =$ `function` p_1`->`$\cdots$`function` p_n`->`e. Il en va de même pour un `let rec` et pour des déclarations locales. La notation `[`e_1`;` $\dots$ `;`e_n`]` est un raccourci pour e_1`::`$\cdots$`::`e_n`::[]`.

2 Analyse sémantique

2.1 Types et environnements de typage

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

$$\tau ::= \texttt{unit} \mid \texttt{string} \mid \texttt{bool} \mid \texttt{int} \mid \alpha \mid \tau \to \tau \mid \tau \times \cdots \times \tau \mid \tau\ \texttt{list}$$

Un environnement de typage Γ est une suite de déclarations de la forme $x : \forall\alpha_1, \dots, \alpha_n.\tau$ où $\forall\alpha_1 \dots \alpha_n.\tau$ désigne un *schéma* de type, *i.e.* un type τ où les variables α_i sont généralisées. Un schéma de type peut être réduit à un type si $n = 0$. Par soucis de concision, on utilisera la notation $\bar{\alpha}$ pour désigner un ensemble de variables $\{\alpha_1, \dots, \alpha_n\}$ et de même, on pourra écrire $\forall\bar{\alpha}.\tau$ pour le schéma $\forall\alpha_1, \dots, \alpha_n.\tau$. L'environnement Γ peut être vu comme une fonction partielle et on notera $\Gamma(x)$ le schéma de type associé à x, s'il existe. On note $\texttt{inst}(\forall\alpha_1, \dots, \alpha_n.\tau)$ l'ensemble des types obtenus en substituant dans τ les variables α_i par des types quelconques.

Dans la suite, on utilisera la notation $\Gamma \oplus p : \forall\bar{\alpha}.\tau$ définie récursivement de la manière suivante:

$$\begin{aligned} \Gamma \oplus \_ : \forall\bar{\alpha}.\tau &= \Gamma \\ \Gamma \oplus x : \forall\bar{\alpha}.\tau &= \Gamma, x : \forall\bar{\alpha}.\tau \\ \Gamma \oplus (p_1, \dots, p_n) : \forall\bar{\alpha}.\tau_1 \times \cdots \times \tau_n &= \Gamma \oplus p_1 : \forall\bar{\alpha}.\tau_1 \oplus \cdots \oplus p_n : \forall\bar{\alpha}.\tau_n \end{aligned}$$

Enfin, si $\texttt{vars}(\tau)$ représente l'ensemble des variables de type apparaissant dans le type τ, on note $\texttt{fv}(\forall\alpha_1 \dots \alpha_n.\tau)$ l'ensemble de variables défini par $\texttt{vars}(\tau) \setminus \{\alpha_1, \dots, \alpha_n\}$.

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

$$\begin{aligned}
\texttt{print_int} &: \texttt{int} \to \texttt{unit} \\
\texttt{print_string} &: \texttt{string} \to \texttt{unit} \\
\texttt{print_newline} &: \texttt{unit} \to \texttt{unit} \\
\texttt{read_int} &: \texttt{unit} \to \texttt{int} \\
\texttt{read_line} &: \texttt{unit} \to \texttt{string}
\end{aligned}$$

2.2 Typage des expressions

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

$$\frac{c \text{ constante de type } \tau}{\Gamma \vdash c : \tau} \qquad \frac{\tau \in \texttt{inst}(\forall\alpha.\alpha\ \texttt{list})}{\Gamma \vdash \texttt{[]} : \tau} \qquad \frac{\tau \in \texttt{inst}(\Gamma(x))}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e : \texttt{int}}{\Gamma \vdash \texttt{-}\ e : \texttt{int}} \qquad \frac{\Gamma \vdash e : \texttt{bool}}{\Gamma \vdash \texttt{not}\ e : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int} \quad op \in \{\texttt{=},\ \texttt{<>}, \texttt{<=}, \texttt{>=}, \texttt{<}, \texttt{>}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int} \quad op \in \{\texttt{+},\ \texttt{-},\ \texttt{*}, \texttt{/}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \texttt{bool} \quad op \in \{\texttt{\&\&},\ \texttt{||}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau\ \texttt{list}}{\Gamma \vdash e_1 :: e_2 : \tau\ \texttt{list}} \qquad \frac{\Gamma \vdash e_i : \tau_i}{\Gamma \vdash (e_1, \ldots, e_n) : \tau_1 \times \cdots \times \tau_n}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau'\ \texttt{list} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \oplus p_1 : \tau' \oplus p_2 : \tau'\ \texttt{list} \vdash e_3 : \tau}{\Gamma \vdash \texttt{match}\ e_1\ \texttt{with}\ [] \to e_2 \mid p_1 :: p_2 \to e_3 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \oplus p : \forall\bar{\alpha}.\tau_1 \vdash e_2 : \tau_2 \quad \bar{\alpha} = \texttt{fv}(\tau_1) \setminus \texttt{fv}(\Gamma)}{\Gamma \vdash \texttt{let}\ p\texttt{=}e_1\ \texttt{in}\ e_2 : \tau_2}$$

$$\frac{\Gamma, x : \tau \vdash e_1 : \tau \quad \Gamma, x : \forall\bar{\alpha}.\tau \vdash e_2 : \tau_3 \quad \bar{\alpha} = \texttt{fv}(\tau) \setminus \texttt{fv}(\Gamma) \quad \tau = \tau_1 \to \tau_2}{\Gamma \vdash \texttt{let rec}\ x\texttt{=}e_1\ \texttt{in}\ e_2 : \tau_3}$$

$$\frac{\Gamma \oplus p : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{function}\ p \to e : \tau_1 \to \tau_2}$$

En plus de ces règles de typage, il convient de vérifier que, pour chaque construction impliquant un motif, aucune variable n'apparaît deux fois dans ce motif. Plus précisément, il faut vérifier l'unicité des variables dans

- le motif p dans la construction `function` $p \to \ldots$
- le motif p dans la construction `let` p `=` $\ldots$
- la paire de motifs p_1,p_2 dans la construction `match` e_1 `with` $[] \to \ldots \mid p_1 :: p_2 \to \ldots$

2.3 Typage des déclarations

On introduit le jugement $\Gamma_1 \vdash d \Rightarrow \Gamma_2$ pour le typage des déclarations. On le définit par les règles d'inférence suivantes :

$$\frac{\Gamma \vdash e : \tau \quad \bar{\alpha} = \mathtt{fv}(\tau) \setminus \mathtt{fv}(\Gamma)}{\Gamma \vdash \mathtt{let}\ p{=}e \Rightarrow \Gamma \oplus p : \forall \bar{\alpha}.\tau}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau \quad \bar{\alpha} = \mathtt{fv}(\tau) \setminus \mathtt{fv}(\Gamma) \quad \tau = \tau_1 \to \tau_2}{\Gamma \vdash \mathtt{let\ rec}\ x{=}e \Rightarrow \Gamma, x : \forall \bar{\alpha}.\tau}$$

2.4 Typage des fichiers

On introduit finalement le jugement $\Gamma \vdash_f d_1 \cdots d_n$ signifiant « dans l'environnement Γ le fichier constitué par la suite de déclarations d_1, …, d_n est bien formé ». Le typage d'un fichier consiste à typer successivement les déclarations dans le contexte étendu par chaque nouvelle déclaration, d'où les règles :

$$\frac{}{\Gamma \vdash_f \emptyset} \qquad \frac{\Gamma \vdash d_1 \Rightarrow \Gamma_1 \quad \Gamma_1 \vdash_f d_2 \cdots d_n}{\Gamma \vdash_f d_1\ d_2 \cdots d_n}$$

3 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 d'Objective Caml. En particulier, l'ordre d'évaluation des arguments d'une fonction, des éléments d'une liste ou d'un n-uplet 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 clôtures et les listes.

Le schéma de compilation pourra être le suivant :

- toutes les valeurs Caml contenues dans des variables sont stockées dans un mot de 32 bits ; plus précisément
  - les entiers sont directement représentés par des entiers 32 bits (petite différence avec Caml),
  - les constantes `()`, `false` et `[]` sont représentées par l'entier 0 et la constante `true` par l'entier 1,
  - un n-uplet, une liste (autre que `[]`) ou une valeur fonctionnelle est représenté par un pointeur vers une donnée allouée sur le tas ;
- 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) ;
- une valeur fonctionnelle est représentée par une *clôture*, c'est-à-dire un bloc alloué sur le tas contenant d'une part un pointeur vers le code de la fonction et d'autre part les valeurs des variables libres de cette fonction.

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`, `print_string` 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`, `print_string` et `print_newline` ;
- de respecter la sémantique de ces fonctions (pas de retour-chariot après `print_int` et `print_string` notamment).

4 Travail demandé

Écrire un compilateur, appelons-le petitcaml, acceptant sur sa ligne de commande exactement un fichier Petit Caml (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 (ne donnez pas les fichiers compilés). Quand on se place dans ce répertoire, la commande `make` doit créer votre compilateur, qui sera appelé petitcaml. La commande `make clean` doit effacer tous les fichiers que `make` a engendrés et ne laisser dans le répertoire que les fichiers sources.

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.