Compilation
Première partie, cahier des charges

18 octobre 2001

1 Compilation 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 : `courant-filliatre.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 `camlette`. La commande `make clean` doit effacer tous les fichiers que `make` a engendré et ne laisser dans le répertoire que les fichiers sources.

2 Conventions lexicales de Camlette

2.1 Blancs

Les caractères espace, retour chariot et tabulation sont considérés comme blancs. Ils sont pris en compte uniquement lors de l'analyse lexicale en tant que séparateurs et sont ignorés par la suite. Plus précisément, un *lexème* ne peut chevaucher un blanc : la présence d'un blanc permet donc de forcer la séparation entre deux lexèmes.

Notez que deux lexèmes ne sont pas nécessairement séparés par un blanc. Par exemple 33*21 sera analysé comme une suite de trois lexèmes.

2.2 Commentaires

Les débuts de commentaires sont marqués par le lexème '`(*`' et les fins de commentaires par le lexème '`*)`'. Les parties d'un programme qui sont commentées doivent être traitées de la même façon que les blancs par l'analyseur lexical.

Il est impératif de pouvoir commenter un bout de programme facilement. Plus précisément, si on prend un programme correct, on devra pouvoir commenter toute suite de lexèmes en ajoutant un début de commentaire entre deux lexèmes ainsi qu'une fin de commentaire entre deux lexèmes, plus loin dans le programme. Lorsqu'on fait cela, toute la zone comprise entre ces deux marques de commentaires est considérée comme étant commentée.[1]

Si un programme est lexicalement correct, l'ajout d'un début de commentaire sans ajout d'une fin de commentaire doit le rendre incorrect.

Afin d'avoir la plus grande liberté possible pour écrire du texte entre commentaires, on essaiera d'autoriser d'écrire le plus possible de choses entre une marque de début de commentaire et une marque de fin de commentaire tant que cela ne contrevient pas aux règles précédentes sur les commentaires.

[1] Notez que le langage C ne vérifie pas cette propriété car les commentaires ne peuvent être imbriqués : pour commenter une partie de code contenant des commentaires, il ne suffit donc pas d'insérer un début de commentaire au début de cette partie et une fin de commentaire à la fin.

2.3 Mots clés

Les mots clés sont les suivants : `if then else true false fun let rec in not type external unit bool int float string`

2.4 Identificateurs (*id*)

Les identificateurs sont constitués d'une suite non vide de caractères dont le premier est alphabétique et dont tous les autres sont alphanumériques ou '`_`'. Leur longueur est arbitraire. Les mots clés de Camlette ne peuvent être acceptés comme identificateurs.

2.5 Constantes (*const*)

Les constantes sont les constantes entières, flottantes, chaînes, booléennes et une constante particulière appelée unit.

- Une constante entière est une suite non vide de chiffres.
- Une constante flottante est une suite non vide de chiffres, suivie d'un point, d'une suite éventuellement vide de chiffres, et éventuellement du caractère '`e`' ou du caractère '`E`' suivis par une suite non vide de chiffres éventuellement précédée d'un signe ('`+`' ou '`-`').
- Une constante chaîne se commence et se termine par le caractère '`"`'. Elle contient une suite de séquences, une séquence étant soit un caractère autre que \, retour chariot, tabulation et '`"`', soit une séquence d'échappement. Une séquence d'échappement est '`\\`', '`\n`', '`\t`' ou '`\"`'. Ces séquences d'échappement permettront respectivement de représenter les caractères '\', retour chariot, tabulation et '`"`'.
- Les constantes booléennes sont `true` et `false`.
- La constante unit est notée `()`.

3 Syntaxe de Camlette

La syntaxe que votre compilateur doit accepter est donnée par la grammaire suivante, d'axiome prog :

$$\begin{array}{rcl}
\mathit{prog} & ::= & \epsilon \\
 & | & \mathit{decl}\ \mathit{prog}
\end{array}$$

$$\begin{array}{rcl}
\mathit{decl} & ::= & \texttt{type}\ \mathit{id} \\
 & | & \texttt{external}\ \mathit{id}\ \texttt{:}\ \mathit{simpletype} \\
 & | & \texttt{let}\ \mathit{def}
\end{array}$$

$$\begin{array}{rcl}
\mathit{def} & ::= & \mathit{id}\ \mathit{paramlist}\ \texttt{=}\ \mathit{expr} \\
 & | & \texttt{rec}\ \mathit{id}\ \mathit{paramlist}\ \texttt{:}\ \mathit{type}\ \texttt{=}\ \mathit{expr}
\end{array}$$

$$\begin{array}{rcl}
\mathit{paramlist} & ::= & \mathit{param}\ \mathit{paramlist} \\
 & | & \epsilon
\end{array}$$

$$\mathit{neparamlist} \quad ::= \quad \mathit{param}\ \mathit{paramlist}$$

$$\mathit{param} \quad ::= \quad \texttt{(}\ \mathit{id}\ \texttt{:}\ \mathit{type}\ \texttt{)}$$

$$
\begin{array}{rcl}
\mathit{expr} & ::= & \mathit{const} \\
& | & \mathit{id} \\
& | & \texttt{fun}\ \mathit{neparamlist}\ \texttt{->}\ \mathit{expr} \\
& | & \texttt{let}\ \mathit{def}\ \texttt{in}\ \mathit{expr} \\
& | & \mathit{expr}\ \mathit{expr} \\
& | & \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{expr}\ \texttt{else}\ \mathit{expr} \\
& | & \mathit{expr}\ \mathit{binop}\ \mathit{expr} \\
& | & \mathit{unop}\ \mathit{expr} \\
& | & \texttt{(}\mathit{expr}\texttt{)}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{atomictype} & ::= & \mathit{id} \\
& | & \mathit{typeconst}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{type} & ::= & \mathit{atomictype} \\
& | & \mathit{type}\ \texttt{->}\ \mathit{type} \\
& | & \texttt{(}\mathit{type}\texttt{)}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{simpletype} & ::= & \mathit{atomictype} \\
& | & \mathit{atomictype}\ \texttt{->}\ \mathit{simpletype}
\end{array}
$$

où *const* désigne les constantes, *id* les identificateurs, *typeconst* les types `unit`, `bool`, `int`, `float`, `string`, *unop* les opérateurs unaires '`-`' et '`not`' et *binop* les opérateurs binaires '`+`', '`*`', '`-`', '`/`', '`=`', '`<`', '`<=`', '`>`', '`>=`', '`<>`', '`&&`' et '`||`'.

Les opérateurs arithmétiques sont munis des priorités habituelles, les opérateurs de comparaison doivent avoir une priorité plus faible que ceux-ci et les opérateurs logiques une priorité encore plus faible, avec '`&&`' prioritaire sur '`||`' prioritaire sur `not`. La flèche sur les types associe à droite (comme en Caml). L'application a une priorité plus grande que tous les opérateurs, eux-mêmes prioritaires sur le `if`...`then`...`else`, lui-même prioritaire sur le `let`...`in` et le `fun`....

4 Équivalences sémantiques

On précise ici que les différentes formes syntaxiques suivantes sont absolument équivalentes du point de vue de la sémantique. Il est donc fortement préférable de produire le même arbre de syntaxe abstraite pour ces différentes formes : cela permettra d'une part de diminuer la quantité de travail nécessaire par la suite et d'autre part d'assurer que ces différentes formes seront traitées de la même façon.

$$
\begin{array}{rcl}
\texttt{fun}\ (\mathit{id}{:}\tau)\ \mathit{paramlist}\ \texttt{->}\ \mathit{expr} & \equiv & \texttt{fun}\ (\mathit{id}{:}\tau)\ \texttt{->}\ \texttt{fun}\ \mathit{paramlist}\ \texttt{->}\ \mathit{expr} \\
\texttt{let}\ \mathit{id}\ \mathit{paramlist} = \mathit{expr} & \equiv & \texttt{let}\ \mathit{id} = \texttt{fun}\ \mathit{paramlist}\ \texttt{->}\ \mathit{expr} \\
\texttt{rec}\ \mathit{id}\ (x_1{:}\tau_1)\ \ldots\ (x_n{:}\tau_n) : \tau = \mathit{expr} & \equiv & \texttt{rec}\ \mathit{id}{:}\ \tau_1\texttt{->}\ \ldots\ \texttt{->}\ \tau_n\ \texttt{->}\ \tau = \texttt{fun}\ (x_1{:}\tau_1)\ \ldots\ (x_n{:}\tau_n)\ \texttt{->}\ \mathit{expr}
\end{array}
$$

Par ailleurs, les parenthèses ne servent qu'à grouper les termes. Une expression de la forme `(`*expr*`)` doit donc être analysée de la même façon que l'expression *expr*.

5 Invocation du compilateur

La commande `camlette` [*options*] $\mathit{fichier}_1 \ldots \mathit{fichier}_n$ compile les fichiers $\mathit{fichier}_1, \ldots, \mathit{fichier}_n$. La valeur de retour de la commande est

- 0 si tout s'est bien passé ;
- 1 si le fichier donné en entrée est incorrect ;
- 2 si vous détectez une erreur dans votre compilateur (par exemple, vous arrivez sur un cas qui n'aurait pas dû être possible).

Notez que le code de retour 2 est le code de sortie d'un programme Caml qui lance une exception et ne la rattrape pas.

Les options possibles doivent comprendre au moins :

-help : affiche un message décrivant la façon d'invoquer le compilateur.

-about : affiche les noms des auteurs, et éventuellement le numéro de version du compilateur.

-no-parse : s'arrête avant l'analyse syntaxique. En d'autres termes, effectue uniquement l'analyse lexicale.

-no-sem : s'arrête avant l'analyse sémantique. Autrement dit, effectue uniquement l'analyse lexicale et l'analyse syntaxique.

Dans la première partie, `camlette` échouera si on ne se restreint pas à faire l'analyse lexicale ou à l'analyse syntaxique et devra calculer l'arbre de syntaxe abstraite . `camlette` se contentera de faire selon les cas l'analyse lexicale et syntaxique de chacun des fichiers qui lui sont donnés et de calculer l'arbre de syntaxe abstraite de ces fichiers ou simplement l'analyse lexicale (en mode `-no-parse`).

Vous pouvez de façon optionnelle ajouter une option **-debug** qui fera afficher diverses informations de déverminage à votre compilateur.

En cas d'erreur lexicale, `camlette` doit afficher un message sur la sortie standard d'erreur indiquant la position dans le fichier original de l'erreur en respectant la syntaxe utilisée dans l'exemple suivant :

```
File "toto.ml", line 10, characters 24-29 : Lexical error.
```

De même, en cas d'erreur syntaxique on écrira :

```
File "titi.ml", line 277, characters 10-96 : Syntax error.
```