Compilation d'IMP en bytecode

Le but de ce TP est de programmer la partie compilation vers machine abstraite du langage IMP, un petit langage impératif. Les principaux composants du programme vous sont donnés : analyseurs lexical et syntaxique, boucles principales (compilation et exécution), interprète de bytecode, ainsi que quelques éléments de la compilation.

Les fichiers desquels vous devez partir sont dans dans le répertoire `TDs/10` de la page du cours et sont les suivants :

- `Makefile` : on ne le présente plus
- `compile.ml(i)` : le squelette de la compilation des expressions d'IMP
- `impast.ml` : le type des arbres de syntaxe abstraite, et leur imprimeur
- `implex.mll` : l'analyseur lexical
- `impparse.mly` : l'analyseur syntaxique
- `mainCompile.ml` : la boucle principale de compilation
- `mainRun.ml` : la boucle principale de l'interprète de bytecode
- `printByteCode.ml` : les fonctions permettant d'afficher du bytecode textuellement
- `vmBytecode.mli` : les définitions de types liées au bytecode
- `vmExec.ml` : la mécanique d'exécution de la machine virtuelle
- `*.imp` : quelques exemples de programmes en IMP

Compiler et recompiler

La commande `make` permet de reconstruire 2 exécutables après chacune de vos modifications :

- `impc` : le compilateur du langage IMP vers du bytecode
- `imprun` : l'interprète de fichiers de bytecode

Note : La toute première fois, après récupération des fichiers, vous devrez créer un fichier (vide) `.depend` dans le répertoire où vous aurez mis les fichiers. Ceci permettra une gestion automatisée des dépendances de compilation (quoi recompiler quand un fichier a été modifié). Vous invoquerez alors `make depend` afin que ces dépendances soient calculées initialement. Lorsque vous modifierez vos fichiers, si vous utilisez une fonction d'une unité de compilation non encore utilisée dans un fichier, la compilation pourra échouer car les dépendances entre fichiers ne seront peut-être pas à jour (en fait, c'est très peu probable car le squelette impose déjà les dépendances nécessaires). Dans ce cas, re-invoquez `make depend`.

Le langage IMP

Vous pouvez consulter le fichier `impparse.mly` pour une description détaillée de la syntaxe d'IMP. Toutefois, quelques exemples ci-dessous devraient vous permettre de vous faire une idée rapide de sa syntaxe.

Il comporte les expressions arithmétiques, le `if`, le `while`, l'affectation `:=`, la séquence et les appels de fonctions `f (x, 3, "Foo")`. Il est possible de définir des variables globales et des variables locales aux fonctions (en tout début du corps). Les instructions autres que le `if` et le `while` sont terminées par un point-virgule. Une «fonction» `print` prenant un nombre quelconque d'arguments de n'importe quels types est prédéfinie dans le langage.

Un programme doit obligatoirement comporter au moins une fonction `main` qui ne prend pas d'arguments.

Il n'y a pas de spécification des types ni d'inférence : un programme incorrectement typé échouera à l'exécution.

```
fact (n)
begin
if n == 0 then
  return (1) ;
  else
  return (n * fact (n - 1)) ;
  endif
end

main ()
begin
  print (fact (5), "\n") ;
end
```

```
var cinq ;

fact (n)
begin
var res ;
res := 1 ;
while n > 0 do
  res := res * n ;
  n := n - 1 ;
  done
 return (res) ;
 end

main ()
begin
  cinq := 5 ;
  print (fact (cinq), "\n") ;
end
```

La machine virtuelle

L'exécution des instructions de la machine virtuelle est décrite par la fonction de transition ci-dessous montrant l'évolution de l'état de la machine suite à l'exécution de chaque instruction. L'état de la machine est défini par (r, c, s, e, m) avec :

— r : registre,
— c : code à exécuter (liste d'instructions),
— s : la pile (liste de valeurs),
— e : l'environnement (liste d'adresses),
— m : la mémoire (tableau de valeurs).

Vous remarquerez en particulier que l'environnement est désormais « global » et fait partie de l'état de la machine virtuelle. En effet, dans notre langage impératif il n'y a plus de notion de fermeture (ce qui va grandement simplifier la compilation par rapport au TD précédent).

état		état suivant
$(r, \texttt{Loadi}(n) :: c, s, e, m)$	$\Rightarrow$	(n, c, s, e, m)
$(r, \texttt{Loadb}(b) :: c, s, e, m)$	$\Rightarrow$	(b, c, s, e, m)
$(i, \texttt{Plus} :: c, j :: p, e, m)$	$\Rightarrow$	$(\overline{j+i}, c, p, m)$
$(i, \texttt{Sub} :: c, j :: p, e, m)$	$\Rightarrow$	$(\overline{j-i}, c, p, m)$
$(i, \texttt{Equal} :: c, j :: p, e, m)$	$\Rightarrow$	$(\overline{j=i}, c, p, m)$
$(i, \texttt{Lt} :: c, j :: p, e, m)$	$\Rightarrow$	$(\overline{j<i}, c, p, m)$
$(\mathsf{true}, \texttt{Branch}(c_1, \_) :: c, s, e, m)$	$\Rightarrow$	$(\mathsf{true}, c_1, s, e, m)$
$(\mathsf{false}, \texttt{Branch}(\_, c_2) :: c, s, e, m)$	$\Rightarrow$	$(\mathsf{false}, , c_2, s, e, m)$
$(\mathsf{true}, \texttt{Loop}(c_1, c_2) :: c, s, e, m)$	$\Rightarrow$	$(\mathsf{true}, c_2 :: c_1 :: \texttt{Loop}(c_1, c_2) :: c, s, e, m)$
$(\mathsf{false}, \texttt{Loop}(c_1, c_2) :: c, s, e, m)$	$\Rightarrow$	$(\mathsf{false}, c, s, e, m)$
$(r, \texttt{Push} :: c, s, e, m)$	$\Rightarrow$	$(r, c, r :: s, e, m)$
$(\_, \texttt{Pop} :: c, v :: s, e, m)$	$\Rightarrow$	(v, c, s, e, m)
$(i, \texttt{Mkblock} :: c, s, e, m)$	$\Rightarrow$	$(@, c, s, e, m)$
	$\Rightarrow$	où @ est l'adresse retournée par
	$\Rightarrow$	l'allocateur pour un bloc de taille i
$(@, \texttt{Envext} :: c, s, e, m)$	$\Rightarrow$	$(\overline{@}, c, s, \overline{@} :: e, m)$
$(\_, \texttt{Read}(n) :: c, s, e, m)$	$\Rightarrow$	$(\overline{m[e(n)]}, c, s, e, m)$
$(r, \texttt{Assign(n)} :: c, s, e, m)$	$\Rightarrow$	$(r, c, s, e, \overline{m[e(n)] \leftarrow r})$
$(r, \texttt{Print} :: c, s, e, m)$	$\Rightarrow$	(r, c, s, e, m)
	$\Rightarrow$	et provoque l'affichage de r
$(\_, \texttt{Pushenv} :: c, s, e, m)$	$\Rightarrow$	$(\_, c, e :: s, e, m)$
$(\_, \texttt{Popenv} :: c, e' :: s, e, m)$	$\Rightarrow$	$(\_, c, s, e', m)$
$(\_, \texttt{Call(f)} :: c, s, e, m)$	$\Rightarrow$	$(\_, f_c, c :: s, e, m)$
	$\Rightarrow$	où f_c est le code de f
	$\Rightarrow$	trouvé dans le format de «l'exécutable»
$(\_, \texttt{Return} :: c, s, e, m)$	$\Rightarrow$	$(\_, [\,], s, e, m)$
$(r, [\,], c :: s, e, m)$	$\Rightarrow$	(r, c, s, e, m)
$(r, [\,], [\,], e, m)$	$\Rightarrow$	Fin d'exécution : succès
Autres configurations	$\Rightarrow$	Fin d'exécution : erreur

1 Travail à faire

Vous devez étendre le fichier `compile.ml` afin de générer le bytecode pour les instructions non gérées. Certains concepts sont similaires au TP précédent dans lequel vous avez implémenté la compilation de PCF. La nouveauté est que le schéma de compilation ne vous est pas donné contrairement au TP précédent où il était décrit dans le polycopié.

1.1 Échauffement

Q1 Implémentez la compilation de `Print` dans la fonction `compile_instr` du fichier `compile.ml`. Cette construction prédéfinie d'IMP prend un nombre quelconque d'arguments, représentés sous forme d'une liste (c.f. `impast.ml`). Comme son nom l'indique, elle permet d'afficher les valeurs des expressions qu'elle reçoit en paramètre.

Q2 Implémentez la séquence (`Seq`) dans la fonction `compile_instr`.

Q3 Testez votre compilateur avec un programme composé d'un `main` n'affichant que des constantes. Par exemple :

```
main ()
begin
  print (1, "Foobar", "\n") ;
  print (45, "\n") ;
end
```

Vous devrez compiler votre programme IMP avec `./impc` et lancer l'exécution du fichier `a.out` généré avec `imprun`.

La mémoire est un tableau de «valeurs» (au sens `VmBytecode.vm_val`). Un allocateur est chargé d'allouer des blocs de «valeurs» (d'une taille spécifiée lors de la demande) lorsque la machine virtuelle exécute l'instruction d'allocation (c.f. plus loin). Cet allocateur vous est donné dans `mem.ml(i)` et n'est pas le sujet de ce TP.

1.2 Un peu moins trivial

Q4 Complétez la génération du code pour les opérateurs binaires dans la fonction `compile_expr`. Le filtrage retournant l'instruction de bytecode associée à chaque opérateur vous est déjà donné pour raccourcir le code à écrire.

Q5 Implémentez l'affectation (`Assign`) dans la fonction `compile_instr`. La fonction `find_pos` qui vous est donnée en début de fichier permet de retrouver dans l'environnement la position d'un identificateur (cf. TD précédent).

Bien qu'IMP permette la déclaration et manipulation de tableaux il ne vous est pas demandé de gérer ces constructions. Vous ne devrez gérer que les variables scalaires. Vous pourrez lire, dans la correction, l'implémentation des tableaux.

Q6 Complétez la fonction `compile_var_decls` qui gère la déclaration d'une liste de variables. Ne considérez pas les tableaux. L'instruction de la machine abstraite permettant d'allouer un «bloc» (zone de donnée en mémoire) `VMI_Mkblock`. Elle s'attend à trouver la taille du bloc dans le registre. Pour allouer **1** référence, la taille sera donc de 1.

Pensez à ce qui se passe au niveau de l'environnement.

Q7 Enfin, implémentez la compilation de l'instruction `while`.

Q8 Vous devez maintenant pouvoir tester votre compilateur avec un programme calculant la factorielle de manière itérative. Le fichier `fact.imp` contient un tel programme mais fait un appel de fonction, construction que vous n'avez pas encore gérée.

1.3 Pour finir, l'application de fonction

Dans IMP, toutes les variables sont mutables. Les arguments d'appels de fonctions sont passés par l'intermédiaire de l'environnement. Lors d'un appel, pour un argument, une variable (référence) est créée en mémoire. L'adresse de cette variable est enregistrée dans l'environnement à destination de la fonction appelée. La valeur de l'expression correspondant à l'argument («paramètre effectif») est stockée dans la référence allouée.

Rappel : l'environnement est global et fait partie de l'état de la machine abstraite : il n'y a plus de notion de fermeture pour les fonctions dans Imp.

Q9 Quel impact cette façon de procéder a-t-elle sur l'environnement lors de l'évaluation des arguments dans la fonction appelante ?

Q10 L'instruction de la machine virtuelle permettant d'effectuer un appel est `VMI_Call`. Que devez-vous faire avant et après en ce qui concerne l'environnement ?

Q11 Complétez la fonction `compile_app` du fichier `compile.ml` afin de gérer le cas de l'application.

1.4 S'il vous reste du temps ...

... implémentez la gestion des tableaux (déclaration, accès en lecture, accès en écriture).