IN 213 Énoncé

Compilation de PCF en bytecode

Introduction

Le but de ce TP est de programmer la partie compilation vers machine abstraite du langage PCF ainsi que la partie exécution de ladite machine abstraite. 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 l'archive de sujet et sont les suivants :

- `pcflex.mll` : l'analyseur lexical
- `pcfast.ml` : le type des arbres de syntaxe abstraite, et leur imprimeur
- `Makefile` : l'incontournable
- `pcfparse.mly` : l'analyseur syntaxique
- `vmBytecode.mli` : les définitions de types liées au bytecode
- `printByteCode.ml(i)` : les fonctions permettant d'afficher du bytecode textuellement
- `mainRun.ml` : la boucle principale de l'interprète de bytecode
- `vmExec.ml(i)` : le squelette (déjà avancé) de la fonction `next_state` qui effectue l'exécution d'une instruction de bytecode dans la machine virtuelle.
- `mainCompile.ml` : la boucle principale de compilation
- `compile.ml(i)` : le squelette de la compilation des expressions de PCF
- `test.pcf` : un exemple de programme PCF pour tester.

Ce TP s'appuie sur les règles vues au cours précédent, plus amplement détaillées dans le polycopié de cours, section 5.5. N'hésitez donc pas à vous y référer.

Compiler et recompiler

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

- `pcfc` : le compilateur du langage PCF vers du bytecode, qui affiche le bytecode généré et le stocke également dans le fichier `a.out`.
- `pcfrun` : 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`.

1 Compilation

Q1 Complétez `compile.ml` afin que la fonction `compile_expr` gère effectivement les constantes.

Q2 Testez avec un programme consistant en l'évaluation d'une constante. Par exemple `5 ;` qui devrait vous générer le bytecode `Loadi 5`.

Q3 Continuez à compléter pour gérer la conditionnelle.

Q4 Testez, par exemple avec le programme `if 5 then 3 else "toto" ;` Remarquez que ce programme est affreusement typé, mais comme on n'a pas de typeur, le compilateur génère quand même du code qui sera incorrect à l'exécution. Le bytecode généré devra ressembler à :

```
Push
Loadi 5
Branch then
  Loadi 3
Branch else
  Loads toto
Branch end
```

Q5 Rajoutez la compilation des opérateurs unaires et binaires.

Q6 Testez le programme `5 - 1 ;`. Le bytecode généré devra ressembler à :

```
Push
Loadi 1
Swap
Loadi 5
Sub
```

Q7 Gérez les fonctions et les `let`-définitions. Pensez à l'équivalence entre les deux.

Q7 Réfléchissez à comment vous pouvez accéder à un identificateur dans l'environnement de la machine virtuelle. Ajoutez l'application et les identificateurs. Rappelez-vous qu'en cours nous avons vu que la notation de De Bruijn correspond aux nombre de lieurs à traverser pour trouver celui d'un identificateur. Cela doit vous donner un indice par rapport à la gestion d'un environnement.

2 Interprète de bytecode

Q8 Complétez `vmExec.ml` afin que la fonction `next_state` gère effectivement toutes les instructions de bytecode. Un certain nombre de cas sont déjà gérés dans le squelette, vous pouvez vous en inspirer. La description des règles d'exécution des instructions présente dans le polycopié est néanmoins suffisante pour achever cette partie.

Q9 Testez votre interprète avec des bytecodes générés par votre compilateur. Par exemple, avec le fichier `test.pcf` fourni, le résultat attendu est 9.

Q10 Testez-le avec le programme mal typé suggéré pour tester la compilation de la conditionnelle.

3 S'il vous reste du temps ...

... enrichissez votre langage à votre gré : traitement des couples, environnement primitif plus riche, définitions globales, définitions récursives *etc.*