Université Paris Cité
L3 Informatique

Machines Virtuelles
Année 2023-2024

TP n°9

Compiler Myrthe vers Java bytecode

Le but de ce TP est de compiler Myrthe vers du Java bytecode (JBC) en format Krakatau (voir TP précédent pour son installation et utilisation) :

github.com/Storyyeller/Krakatau

Sur la page Web du cours vous trouvez une archive avec les fichiers pour démarrer. L'archive contient un analyseur lexical appelé aussi lexeur (écrit en ocamllex) ainsi qu'un analyseur syntaxique appelé aussi parseur (écrit en menhir, qui doit être installé) pour des expressions Myrthe. Il suffit de faire `make` pour obtenir le compilateur `compileMyrthe`. Le compilateur très simple prend un fichier en entrée (donné en commande) et produit sur la sortie standard du JBC en format Krakatau. Par exemple, si le fichier `test` contient l'expression Myrthe `0` en exécutant `compileMyrthe test > Myrthe.j` on obtient le fichier `Myrthe.j` qui contient le JBC suivant :

```
.version 49 0
.class super Myrthe
.super java/lang/Object

.method public static expression : ()I
    .code stack 10 locals 10
   iconst_0
   ireturn
    .end code
.end method
.end class
```

Ce code contient une méthode `expression` qui ne prend pas d'arguments et renvoie un entier qui correspond à la valeur de l'expression. Pour l'instant la taille de la pile et le nombre de variables locales sont fixés à `10`. Pour tester ce code, on peut l'assembler vers un fichier `Myrthe.class` en utilisant `assemble.py` de Krakatau (voir TP précédent). Ensuite on peut par exemple prendre un fichier `MyrtheFull.java` qui contient

```
class MyrtheFull {
    public static void main(String[] args) {
        {System.out.println(Myrthe.expression());}
    }
}
```

On le compile avec `javac` et on exécute `java MyrtheFull` qui fait appel à `Myrthe.class`.
Le lexeur et le parseur traitent déjà tout Myrthe (voir la fin de `main.ml` si voulez les tester). Dans ce qui suit vous n'avez donc pas besoin de les modifier. Par contre pour l'instant le compilateur ne sait traiter que les expressions composées de valeurs entières, de l'opération de successeur, de l'addition et de l'égalité comme 1+2+3 ou (1+2)+(++3)=3.

1 Compilation des expressions de Myrthe (sans variables)

Exercice 1. Regardez les fichiers fournis (hormis lexeur et parseur) et essayez de comprendre leur fonctionnement.

Exercice 2. Complétez les fichiers `mv.ml` et `compile.ml` pour traiter la multiplication.
Indications : Il faut ajouter un constructeur pour l'instruction de la JVM qui permet de multiplier deux entiers dans `mv.ml` ainsi que compléter les fonctions `instrlabel_to_string` et `compil` dans `compile.ml`.

Exercice 3. Complétez les fichiers `mv.ml compile.ml` pour traiter les autres constructeurs de Myrthe (pour `<=`, `not`, `||`, `&&`, `if...then...else...`) sauf les variables et leur déclarations (`let ...=... in ...`).

Exercice 4. Le compilateur produit pour l'instant automatiquement du JBC où la taille de la pile est fixée à `10`. Donnez une expression pour laquelle cela pose problème. Modifiez le compilateur de sorte qu'il calcule automatiquement la taille de la pile nécessaire et l'écrit au bon endroit.

2 Compilation des expressions de Myrthe (avec variables)

Comment compiler les expressions avec variables comme

```
let x = 3 in let y = 4 in let x = 2 in x+y+x    ?
```

D'abord pour avoir des variables uniques on peut transformer une expression de Myrthe vers une expression où chaque variable est définit une seule fois (mais utiliser potentiellement plusieurs fois). Par exemple, on transforme l'expression ci-dessus en

```
let x0 = 3 in let x1 = 4 in let x2 = 2 in x2+x1+x2
```

Ensuite, l'idée est d'utiliser une variable locale de la JVM pour chaque variable de Myrthe.

Exercice 5. Écrivez une fonction OCAML qui prend une expression de Myrthe en entrée et qui produit une expression où chaque variable définie est unique. Les variables doivent s'appeler `x0`, `x1`, etc.

Exercice 6. Complétez le compilateur pour traiter les expressions de Myrthe avec variables.
Indication : La définition d'une variable dans un `let` se traduit par un `istore`, `istore_0`, `istore_1`, etc. dans le JBC (voir la liste des instructions JVM) et l'utilisation d'une variable par un `iload`, `iload_0`, etc.

Exercice 7. Modifiez le compilateur pour remplacer le nombre de variables locales (par défaut `10`) par la valeur exacte.