University of
St Andrews

The School of Computer Science

CS4201

Prog. Lang. Design & Implementation

2015/16

Practical 1:

A compiler to JVM Bytecode

ID: 150022355

Submitted: 01/10/2015

Tutors: Kevin Hammond

Short Report

The following practical was to try and define AST datatypes for a Skel concrete syntax language and then compile small programs into JVM bytecode. The experience was incredibly beneficial and thoroughly enjoyable. To begin with, I wanted to push myself out of my comfort zone. Therefore, I decided to code the AST / compiler using Haskell. This was a huge risk, as I had never used Haskell or had I even studied compiler design before. Having said that, at the same time it gave me a new perspective to build upon and gave me greater insight into programming language design. The downside to this though, was that I did not know enough Haskell to take the compiler further and re-evaluate the initial Skel language design. Therefore, the aim was to accumulate and investigate as much as possible in the process.

The Abstract Syntax Tree

Haskell provided an approach to programming AST's which seemed more productive than any other language. Data sets can be assigned using *generalised algebraic datatypes*. For each element in the Skel language provided, the compiler looks like the following:

```
data Struct =     StructExp                Exprs
       | StructComp                        Structs Structs
       | StructIt                          Int Structs
         deriving Show

data Exprs = Expression                    Expr
       | ExprSeperator                     Expr Exprs
         deriving Show

data Expr = ExprInt                        Int
       | ExprString                        String
       | ExprBool                          Bool
       | ExprAssignment                    Id Exprs
       | ExprRaise                         Id Exprs
       | ExprCatch                         Exprs Id Id Exprs
       | ExprOp                            Exprs Op Exprs
       | ExprGroup                         Exprs
         deriving Show
```

Within the same *compiler.hs* document, there are a number of small abstract syntax expressions which use the generalised algerbriac datatypes to compose small programs:

```
printStringInts = Program "printStringInts" ( Par ( ParId "ExprSeperator" (
Struct ( StructExp ( ExprSeperator (  ( ExprString "Hello World!" )   )  (
Expression ( ExprInt 42 ) )  )  )  ) ) )

printStringBool = Program "printStringBool" ( Par ( ParId "ExprSeperator" (
Struct ( StructExp ( ExprSeperator (  ( ExprString "Hello World!" )   )  (
Expression ( ExprBool True ) )  )  )  ) ) )
```

The document is then compiled and depending on what program is selected the datatypes will link together and the `./compiler` command should return the chosen programs expression. Lastly, the next step is to convert the program expression into JVM bytecode. A lot of research was conducted for this as it seems converting Haskell to JVM is not as simple as first assumed. After some debating, my lack of Haskell knowledge did not allow me to undertake any substantial functional programming to convert the expressions into JVM. I was unable to link the chain together and therefore when using Jasmin (an assembler for the JVM) – it seemed as though manually creating the .j files for JVM conversion was the only route. As a result, a number of test programs were conducted. For example, the following `returnBoolFalse` program in JVM bytecode looks like the following:

```
.class public returnBoolFalse
.super java/lang/Object

; standard initializer
; default constructor

.method public <init>()V
   aload_0 ; push this
   invokespecial java/lang/Object/<init>()V ; call super
   return
.end method

.method public static main([Ljava/lang/String;)V

   ; allocate stack big enough to hold 1 item
   .limit stack 2
   .limit locals 1

   ; push java.lang.System.out (type PrintStream)
   getstatic java/lang/System/out Ljava/io/PrintStream;
   ; push int to be printed
   ; 0 = false, 1 = true
   ldc 0
   ; invoke println
   invokevirtual java/io/PrintStream/println(Z)V ; bool print (Z)
   ; terminate main
   return

.end method
```

Which returns the following result to the console:

```
8afbe368:compiler homefolder$ java returnBoolFalse

false
```

The following program `returnBoolTrue` when converted to JVM bytecode returns the expression Boolean value of true:

```
8afbe368:compiler homefolder$ java returnBoolTrue

true
```

The following program `printString` when converted to JVM bytecode returns the expression of a string containing “Hello World!”:

```
8afbe368:compiler homefolder$ java printString

Hello World!
```

The following program `opAddInts` when converted to JVM bytecode returns the operation of adding two integers together “10 + 5”:

```
8afbe368:compiler homefolder$ java opAddInts

15
```

The following program `opMinusInts` when converted to JVM bytecode returns the operation of subtracting an integer from another “100 - 35”:

```
8afbe368:compiler homefolder$ java opMinusInts

65
```

The following program `opMultiplyInts` when converted to JVM bytecode returns the operation of multiplies two integers together “235 * 19”:

```
8afbe368:compiler homefolder$ java opMultiplyInts

4465
```

The following program `opDivideInts` when converted to JVM bytecode returns the operation of divides two integers “345 * 15”:

```
8afbe368:compiler homefolder$ java opDivideInts

23
```

The following program `printStringBool` when converted to JVM bytecode returns a string and a Boolean:

```
8afbe368:compiler homefolder$ java printStringBool

Hello World!
true
```

The following program `printStringInts` when converted to JVM bytecode returns both an integer and string:

```
8afbe368:compiler homefolder$ java printStringBool

Hello World!
42
```

Evaluation

The practical presented a number of difficulties. One of them was when trying to compose a program using multiple expressions. I found this difficult and from looking back, I needed to make one or two design decisions in order to compile multiple expressions. In regards to this, I feel as though my lack of experience let me down. If I had been more aware and more accustomed to breaking down the components of the language – perhaps I would have been able to identify the design choices sooner. This would have enabled me to do further tests on both grouping and raising exceptions. Another issue was trying to decide on the functionalities of both the Parallel Pipeline and Task Farm syntax and what it was they were meant to do. However, I am happy with the attempt and the practical introduced important methods and concepts.