CS4215 Programming Language Implementation

Lab task for Week 08
Compiler and Virtual Machine for simPL

1. In this lab, you will see how “easy” is it to build a compiler for a simple but powerful simPL programming language. You will also see how to design (and implement) a virtual machine with the ability to handle functions as first-class values!

2. Download a folder named lab07.zip from IVLE workbin. The folder contains the following:

   (a) `sPL.ml` which contains the AST of simPL language.

   (b) `sPLc.ml` which contains the AST of a core internal language for simPL.

   (c) You are to replace the type checker `sPL_type.ml` with the version you are implementing for lab06. We will give you our version, after the deadline of lab06 has expired.

   (d) `*.spl` which contains some test examples.

   (e) An OCamlBuild makefile where `make lab07` will allow you to compile multiple files to produce a compiler, named `splc` and its virtual machine `svm`.

   (f) `test7.sh` which is a script to test your compiler and virtual machine against given some test examples and output the results in `out.sp*`.

   (g) `diff7.sh` which is a script to compare your outcome against our expected answers in `test/ref7.out.s*`

3. Deadline for Lab07 is 20th March (Mon) 7pm.

4. The compiler is currently able to generate virtual machine code for primitive operations, non-recursive functions and variables. Your main task is to complete it to handle *conditional*, *application* and *recursive function.* Initially, you shall focus on generating the codes that uses symbolic addresses.

5. You can incrementally complete the various features of the compiler in the `sPL_compile.ml` module. To help you in this task, we have provided some simple examples to guide you on the expected behaviour of the compiler. As a start you may perform the following compilation:

```
loris@lab07$ cat t1.spl
(2 * 3) + 4
loris@lab07$ ./splc t1.spl
Loading sPL program ..
  (2 * 3) + 4
  as +[*[Int(2),Int(3)],Int(4)]
TYPE CHECKING program ..
 ==> inferred type Int
TRANSFORMING ==> +[*[Int(2),Int(3)],Int(4)]
COMPILING ==> t1.svm
[LDCI 2,LDCI 3,TIMES,LDCI 4,PLUS,DONE]
LINKING ==>
[LDCI 2,LDCI 3,TIMES,LDCI 4,PLUS,DONE]
```

This simple example successfully compiles and also perform linking which converts the symbolic addresses to integer form. In this case, there isn't any symbolic addresses. Hence, the code remains unchanged.

Another successful example is the following:

```
loris@lab07$ cat t2.spl
fun {int -> int -> int} x y -> y - 3 + x end
loris@lab07$ ./splc t2.spl
Loading sPL program ..
  fun {int -> int -> int} x y -> y - 3 + x end
  as fun {Int->Int->Int} x y -> +[-[Var(y),Int(3)],Var(x)] end
TYPE CHECKING program ..
 ==> inferred type Int->Int->Int
TRANSFORMING ==> fun {Int->Int->Int} x y -> +[-[Var(y),Int(3)],Var(x)] end
COMPILING ==> t2.svm
[LDF([],2,label_0),DONE,
label_0:,LD (y,1),LDCI 3,MINUS,LD (x,0),PLUS,RTN]
LINKING ==>
[LDF([],2,2),DONE,
2:,LD (y,1),LDCI 3,MINUS,LD (x,0),PLUS,RTN]
```

This example manages to compile a two-argument function and placed its code at symbolic address `label_0`. Note that the two parameters, `x` and `y` are placed at locations `0` and `1` respectively at the current environment of the method. The instruction `LDF([],2,label_0)` builds a closure with arity 2 that denotes our function. The linking phase changes `label_0` to address 2.

The next example involves conditional construct that has not been implemented yet.

```
loris@lab07$ cat t3.spl
```

```
fun {int -> bool} x ->
   if x>5 then true else false end
end
loris@lab07$./splc t3.spl
Loading sPL program ..
  fun {int -> bool} x ->    if x>5 then true else false endend
  as fun {Int->Bool} x -> if >[Var(x),Int(5)]
         then Bool(true) else Bool(false) end
TYPE CHECKING program ..
 ==> inferred type Int->Bool
TRANSFORMING ==> fun {Int->Bool} x -> if >[Var(x),Int(5)]
         then Bool(true) else Bool(false) end
COMPILING ==> t3.svm
Fatal error: exception Failure("TO BE IMPLEMENTED")
```

Hence, we fail in the compilation stage. Your first task is to make sure that the compiler generates correct code for conditional. Once you have done that, you can proceed to `t4.spl` (to handle application), and `t5.spl` `t6.spl` to generate codes for creating recursive methods.

6. Once this is done, perform `test7.sh` and `diff7.sh` to see if your compiler works correctly for our set of test programs. Our congratulations to you for completing a non-trivial compiler!