Lab 1

DD2481 - Principles of Programming Languages

27 March 2018

The goal of this lab is the implementation of an interpreter for the simple language seen in the lecture. You are provided with a code template; below we explain the parts of the template that you need to be familiar with, however, we recommend that you spend some time exploring also other parts of the template not mentioned below. (For reference, we will refer to different files in the source distribution; in each case we omit the common prefix "src/main/scala/simpret/".)

The syntax and small-step evaluation rules follow.

Syntax:

t	$::=$		*terms* :
	$\mid$	true	*constant true*
	$\mid$	false	*constant false*
	$\mid$	if t then t else t	*conditional*
	$\mid$	c	*integer constant*
	$\mid$	iszero t	*zero test*
	$\mid$	$t + t$	*integer addition*
	$\mid$	x	*variable*
	$\mid$	x := t	*assignment*
	$\mid$	t ; t	*sequence*

v	$::=$		*values* :
	$\mid$	true	*true value*
	$\mid$	false	*false value*
	$\mid$	c	*integer value*

$(c \in \{\ldots, -2, -1, 0, 1, 2, \ldots\})$

$(x \in \{\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{y}_1, \mathtt{y}_2, \ldots, \mathtt{z}_1, \mathtt{z}_2, \ldots, \ldots\})$

Evaluation:

E-IfTrue
$$\mathsf{if\ true\ then}\ t_2\ \mathsf{else}\ t_3 \mid \mu \rightarrow t_2 \mid \mu$$

E-IfFalse
$$\mathsf{if\ false\ then}\ t_2\ \mathsf{else}\ t_3 \mid \mu \rightarrow t_3 \mid \mu$$

E-If
$$\frac{t_1 \mid \mu \rightarrow t_1' \mid \mu'}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \mid \mu \rightarrow \mathsf{if}\ t_1'\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \mid \mu'}$$

E-IsZeroZero
$$\mathsf{iszero}\ 0 \mid \mu \rightarrow \mathsf{true} \mid \mu$$

E-IsZeroNonZero
$$\frac{c \neq 0}{\mathsf{iszero}\ c \mid \mu \rightarrow \mathsf{false} \mid \mu}$$

E-IsZero
$$\frac{t_1 \mid \mu \rightarrow t_1' \mid \mu'}{\mathsf{iszero}\ t_1 \mid \mu \rightarrow \mathsf{iszero}\ t_1' \mid \mu'}$$

E-Add
$$c_1 + c_2 \mid \mu \rightarrow c_1\ \mathcal{I}(+)\ c_2 \mid \mu$$

E-AddRight
$$\frac{t_2 \mid \mu \rightarrow t_2' \mid \mu'}{c_1 + t_2 \mid \mu \rightarrow c_1 + t_2' \mid \mu'}$$

E-AddLeft
$$\frac{t_1 \mid \mu \rightarrow t_1' \mid \mu'}{t_1 + t_2 \mid \mu \rightarrow t_1' + t_2 \mid \mu'}$$

E-Deref
$$\frac{x \in dom(\mu) \quad \mu(x) = v}{x \mid \mu \rightarrow v \mid \mu}$$

E-AssignVal
$$x\ \mathsf{:=}\ v \mid \mu \rightarrow v \mid [x \mapsto v]\mu$$

E-Assign
$$\frac{t_1 \mid \mu \rightarrow t_1' \mid \mu'}{x\ \mathsf{:=}\ t_1 \mid \mu \rightarrow x\ \mathsf{:=}\ t_1' \mid \mu'}$$

E-Seq1
$$v\ ;\ t_1 \mid \mu \rightarrow t_1 \mid \mu$$

E-Seq2
$$\frac{t_1 \mid \mu \rightarrow t_1' \mid \mu'}{t_1\ ;\ t_2 \mid \mu \rightarrow t_1'\ ;\ t_2 \mid \mu'}$$

The interpreter should be based directly on the small-step evaluation rules.

In order to be able to parse source code, we are providing you with a code template that includes a fully-functioning and complete lexer and parser (the implementation uses an approach called parser combinators, provided by the Scala library at `https://github.com/scala/scala-parser-combinators`). You do not have to work on these parts at all. The result of parsing a program is an abstract syntax tree (AST). The AST, which is also provided to you, is implemented in Scala as follows (file "parser/AST.scala"):

```
trait AST extends Positional
case class Variable(id: String) extends AST
case class BoolLit(b: Boolean) extends AST
case class IntLit(i: Int) extends AST
case class CondExp(c: AST, e1: AST, e2: AST) extends AST
case class IsZeroExp(e: AST) extends AST
case class PlusExp(e1: AST, e2: AST) extends AST
case class AssignExp(id: String, e: AST) extends AST
case class SeqExp(e1: AST, e2: AST) extends AST
```

Your task is to implement the `step` method of the `Interpreter` singleton object in file "interpreter/Interpreter.scala":

```
def step(x: AST, store: Map[String, AST])
  : Option[(AST, Map[String, AST])] = {
  // TODO: your implementation goes here
}
```

As you can see, the store, which is used to keep track of the current values of variables, is implemented using a Scala Map (basically, a hash map). The two main operations of a map are `get` and `+`. For example, `store.get(id)` looks up the value for identifier `id` in the store and returns a Scala option (similar to the `Optional` type in Java). On the other hand, `store + (id -> e)` creates a new store which is equal to `store`, except that `id` is mapped to `e`. See also the API documentation for `Map`: `http://www.scala-lang.org/api/2.12.4/scala/collection/immutable/Map.html`

In order to do one (small) step of computation, the idea is to match on the AST parameter "x":

```
x match {
  case Variable(id) => ...
  case PlusExp(e1, e2) if !isvalue(e1) => ...
  ...
}
```

For developing and testing your solution you can use the Main object (file "Main.scala").

The project is built using sbt, the most widely used build tool in Scala: `https://www.scala-sbt.org/` Make sure to familiarize yourself with sbt, so that you are able to compile and run the code template. The guide "Getting Started with sbt" is a good starting point:
`https://www.scala-sbt.org/1.x/docs/Getting-Started.html`

You can build the project by invoking `sbt` in the root directory of the project:

```
$ sbt
[info] Loading project definition from ...
[info] Loading settings from build.sbt ...
[info] Set current project to simpret (in build file:/...
[info] sbt server started at local:///...
sbt:simpret>
```

You get to an input prompt where you can enter sbt commands, such as `compile`, for compiling the source of the current project, and `run`, for running the main object. Make sure that `compile` works. (Compiling will output JVM class files into the directory "lab1/target/scala-2.12/classes".) Importantly, you can pass arguments to the `run` command. The code template comes with a main object where you can provide the file name of a source file that contains an expression to be interpreted.

For example, put the following expression into a file "test.sint":

```
if true then 3 else 4
```

Then, at the sbt prompt, enter `run test.sint` and press enter. You should see output similar to the following:

```
sbt:simpret> run test.sint
[info] Packaging /Users/.../lab1/target/scala-2.12/simpret_...
[info] Done packaging.
[info] Running simpret.Main test.sint
============================================================
AST
============================================================
if true then
  3
else
  4
============================================================
Evaluation stuck!
Stuck at
==================
if true then
  3
else
```

```
  4
==================
[success] Total time: 1 s, completed Mar 27, 2018 12:35:13 PM
```

It says that evaluation is stuck, because the interpreter has not been implemented, yet. You also see that it provides a pretty-printed rendering of the AST.

Good luck!