Victoria University of Wellington
School of Engineering and Computer Science

SWEN221: Software Development

Lab 7 — Simple Lisp (worth ≈1.3% of overall mark)

This lab is an exercise in program comprehension and testing. By the end of the lab, the tutor should have given you a grade based on your performance. **Be sure that you have been given a grade before you leave, since attendance at labs is mandatory**.

NOTE: it is also recommended that you submit your final lab code via the online submission system, which can be found on the SWEN221 homepage. This is useful in case your lab grade is lost, or you believe you have been given the incorrect grade.

Simple Lisp

Simple Lisp is an interpreter for a subset of the Lisp language. You can find information on Lisp in the Appendix of this handout, **but you don't need to be able to program in Lisp for this assignment**. Simple Lisp is a small program but probably larger than most programs you have so far encountered. There is no documentation, , very few tests, and it is poorly commented — much like most of the code you will encounter in real life.

Understand it. Spend some time understanding the code. You won't get marks for this, but spending some time reading and understanding will set you up for the following questions. You can ignore the `Parser` and `Lexer` classes.

1 Warm-up test

In this question, you will be performing unit tests and evaluating code coverage for the `type_check` method in the class `Interpreter`.

The `type_check` method takes a function name (`fn`), an array of `LispExpr`s (`es`) and a number of `Class`es (`cs`), the method should check that the each `LispExpr` in `es` is an instance of the corresponding `Class`, it should throw an `Error` if this is not the case.

Open the class `org.simplelisp.interpreter.tests.InterpreterTests`, there are a few tests for a few of the methods in the `Interpreter` class. There is currently only one test for the the `type_check` method. By using the Emma tool (see below for an overview), you should be able to see that code coverage for `Interpreter.type_check` is poor (19.5%). Add more tests to `InterpreterTests` achieve 100% coverage of the `type_check` method.

2 White box testing and code coverage

You should write unit tests for the `LispInteger`, `LispVector`, and `InternalFunctions` classes; you should aim to get 100%, 80% and 50% code coverage (using Emma) for each class, respectively.

As an example, we give you tests for `LispNil` in `org.simplelisp.interpreter.tests.LispNilTests`, you should check using Emma that the coverage is 100%.

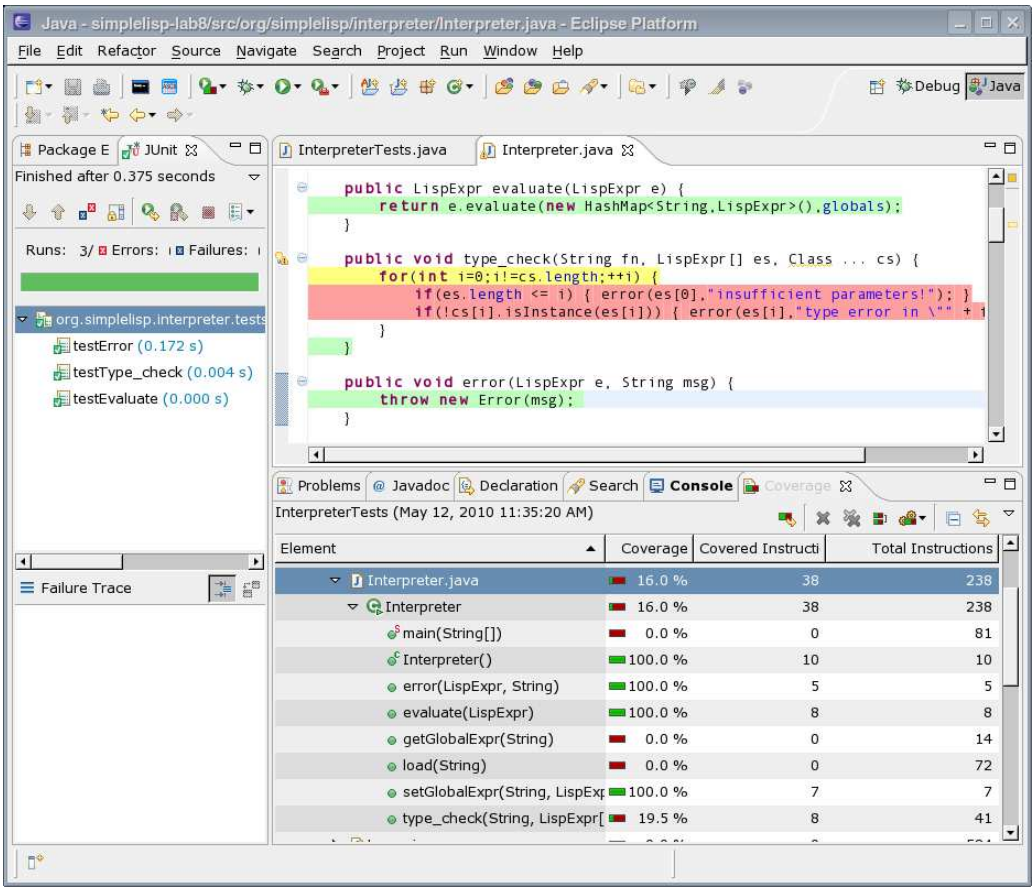

Figure 1: Illustrating the Emma tool being run under Eclipse. Green lines indicate those have been covered by the tests, whilst red lines are those which were not covered. Finally, yellow lines indicates partial coverage which typically occurs for conditionals where e.g. only one side of the conditional is taken.

3 Black box testing

Finally, you should write black box tests for the `evaluate` method of `Interpreter`. To identify appropriate black box tests, you should examine the description of Lisp given in the Appendix. From the description, you should be able to construct numerous tests to cover the basic functionality of the language. **You do not need to do code coverage for this question.**

The `testEvaluate` method in `org.simplelisp.interpreter.tests.InterpreterTests` has a simple test to get you started. You should expand this test method to test that `evaluate` performs correctly on other lisp expressions. You do this by creating an object representation of a lisp expression (using the `Lisp...` classes) and testing that `Interpreter.evaluate` gives the correct result. This is black box testing, so you shouldn't care about how the answer is computed, only that it is correct. Remember to test the boundary cases.

Marking Guide

Each lab is worth just under 1% of your overall mark for SWEN221. The lab should be marked during the lab sessions, according to the following grade scale:

- **0**: Student didn't attend lab.
- **E**: Student did not really participate in the lab.
- **D**: Student's participation was *poor*. For example, he/she made some attempt to work on the lab, but did not complete any activities.
- **C**: Student's participation was *satisfactory*. That is, he/she completed at least one activity (e.g. warm up test).
- **B**: Student's participation was *good*. That is, he/she has completed activities 1 and 2.
- **A**: Student's participation was *excellent*. That is, he/she completed all activities.

A The Emma Tool

Emma is a code coverage tool: for a given test it will tell you how much of your code is covered by that test. Emma is already installed in Eclipse and should be easy to use.

To find the coverage for a test, right click on a JUnit test file, and select "coverage as" and then "JUnit test". The JUnit tests will be run as normal, but in the lower window pane there are test coverage results for the whole program. You should navigate through the package/class hierarchy to find the class or methods you are interested in. You can see pecentage coverage results. By clicking a class or method name, that class will be opened in the main code window and test coverage will be shown graphically: lines highlighted green are covered by the executed tests, lines highlighted red are not covered, lines highlighted yellow are partially covered — that is, some statements on that line are covered and some aren't. The screenshot below shows what this looks like for the supplied tests for the `Interpreter` class.

B The Lisp Programming Language

Lisp was originally created in 1958 by John McCarthy and is one of the oldest high-level programming languages still in use today. The language is characterised by its distinctive fully-parenthesized syntax. Most people find this rather difficult to understand at first, although with time it becomes rather intuitive. Lisp was originally created as a mathematical notation for computer programs based upon the Church's Lambda Calculus and many important ideas in computer science were pioneenered in Lisp.

You do not need to learn Lisp in order to do this lab. However, a basic understanding of the language will probably help. The most important construct in Lisp is the list. This is simply a list of elements separated with spaces and surrounded by brackets. For example, the following is a simple Lisp list:

```
(+ 2 3)
```

Lists are used to construct expressions in Lisp. Lisp uses prefix notation and, hence, the above corresponds to the expression "`2+3`". Evaluating it will give the result "`5`". When Lisp encounters a list to evaluate, the first element of the list determines the function to execute, whilst the remaining elements are its parameters. Therefore, a statement such as

```
print("hello world")
```

from a Java-like language corresponds to the following list in Lisp:

```
(print "Hello World")
```

When Lisp is asked to evaluate a List (such as the above) it will, in the normal course of things, begin by evaluating each element of the list. At this point it will read the first element of the list to determine which function to execute. Consider the following list:

```
(print (+ 1 2))
```

This will print "`3`", not "`(+ 1 2)`", because the list "`(+ 1 2)`" is evaluated before the print function is called. If we want to prevent Lisp from evaluating an element, we can use the special quote operator as follows:

```
(print '(+ 1 2))
```

This prints "`(+ 1 2)`", not "`3`" as before.
Lisp provides several special operators for performing computation. Some useful examples include:

- `(if cond-expr then-expr else-expr)`. This special list type will first evaluate "cond-expr". If this is evaluates something other than "`nil`" (the empty list), Lisp will then evaluate the "then-expr". Otherwise, "else-expr" is evaluated.
- `(car list-expr)`. This returns the "head" of the list. For example, "`(car '(1 2 3))`" returns "`1`".
- `(cdr list-expr)`. This returns the "tail" of the list. For example, "`(cdr '(1 2 3))`" returns "`(2 3)`".
- `(cons expr list-expr)`. This constructs a list by adding "`expr`" to the head of "`list-expr`". So, for example, "`(cons 'A '(B C))`" returns "`(A B C)`".
- `(defun name var-list body ...)`. This defines a new lisp function. The first parameter (i.e. name) determines the name of the function. The second parameter must be a list of variable names (this is the functions parameter list). The remaining elements of the list constitute the body of the function.

There are many other operations provided by Lisp (although, sadly, many are not supported by the Simple Lisp Interpreter at this time).

For those who are interested in learning more about Lisp, much useful information can be found on the internet (or in the library). See for example the Wikipedia entry on Lisp.

B.1 Fibonacci in Lisp

The following Lisp program is provided to illustrate the language. You can run it using the Simple Lisp Interpreter:

```
(defun fibonacci (x)
  ;; compute fibonacci number x
  (if (<= x 1) 1
    (+ (fibonacci (- x 1)) (fibonacci (- x 2)))
    )
  )

(defun series (x)
  (if (= x 0) (print  (fibonacci x))
    (progn
      (series (- x 1))
      (print " ")
      (print (fibonacci x))
      )
    )
)

(defun main ()
  (print "Enter length of series to generate: ")
  (series (- (parse-integer (read-line)) 1))
  (print "\n")
)

(main)
```