UNIVERSITY OF THE WEST INDIES
Department of Computing
CS34Q–Implementation of Interpreters
Sem I, 2015
Lecturer: Dr. Daniel Coore

Project: Checkoff by Monday, December 21, 2015

Instructions: This is the first (and only) group assignment. It requires a substantial amount of work, and therefore needs the full participation of each group member. Please note that there are **two** problems on this question paper.

Collaboration between groups is permitted, but each group's solution must be distinctly its own. Any hint of duplication across groups will be dealt with severely.

Problem 1: SMPL *Interpreter* **[90]** Write an SMPL interpreter in Java. The following language features are optional and attract extra credit if implemented correctly.

- Unicode presentation of character literals. (You must recognise simple character literals enclosed between single quotes though.) **[5]**
- Arithmetic operator symbols embedded within identifiers.
- Tail recursion. You must support recursive procedure calls, but it is optional to prevent tail calls from using extra stack space. **[15]**
- Variable arity procedures. (You must at least support procedures that take a fixed number of arguments.) **[5]**
- Lazy evaluation **[5]**
- Multiple valued expressions and assignment. **[5]**
- Dynamic scoping. **[10]**
- Redefinable built-in procedures. **[10]**

Other optional features mentioned in the language specification will also attract extra credit if implemented. (Do not go overboard, there is a limit to extra credit).

Use JavaCUP and JLex to help you specify the grammar and the tokens of SMPL. You will need to define your own classes to support the specification of the semantics. (Use the files from the previous assignments as a starting point.) You should also use the visitor design pattern to define an appropriate visitor interface for traversing the abstract syntax tree that arises from parsing.

Note that one significant component of the interpreter that you will need to implement yourself is the representation of values in SMPL. In HPL you had only three basic types to contend with (numbers, functions and painters) and since the syntax of the commands provided enough context to infer which type of value was being used, you could constrain the implementation of various aspects of your interpreter appropriately. (For example, you could use separate environment instances to store data of each type.)

Now you will need to plan a little more carefully to ensure that your SMPL values can represent the wide range of possibilities that there are (including functions, numbers, and data structures such as pairs and vectors). You may find useful the approach of creating a top level value class, say `SMPLValue<T>`, and using it as a generic wrapper class to import any Java class (`T`) into the SMPL world of values. Be aware though, that you may have to do some tweaking in order to get subtyping and mixed operations to work properly (e.g. when adding two numbers, the actual operation performed depends on the types of the two numbers). If you need further clarification on this idea, I would be happy to explain.

Problem 2: *Programming in* SMPL **[10]**
Implement the mergesort algorithm in SMPL. Your SMPL mergesort should accept a vector and a comparator (i.e. a function that takes two objects and returns true if the first precedes the second) and return a vector of the same elements in increasing order of precedence, having sorted them using the merge sort algorithm.