UNIVERSITY OF THE WEST INDIES
Department of Computing
COMP3652–Language Processors
Sem I, 2019
Lecturer(s): Prof. Daniel Coore

Project: Schedule checkoff by Friday, Dec. 22, 2019

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 (you are advised to start from a working interpreter for a variant of `arithExp` or one that you might have turned in for Assignment 2).

The following language features are optional and attract extra credit if implemented correctly.

- Identifiers that permit arithmetic operators in them. (You must at least support identifiers, made up of alphanumeric characters, that may start with a number, so long as they contain at least one alphabetic character). **[5]**
- 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]**
- Default parameters. This is not specified in the language specification, but you should use Python syntax for default parameters if you choose to implement this. **[5]**
- Vector comprehension. **[10]**
- Multiple valued expressions and assignment. **[5]**
- Lazy parameter passing. **[10]**
- Call by reference parameter passing. **[10]**
- Redefinable built-in procedures. **[10]**
- Iteration forms (e.g. `for`, `while`, `repeat`). In the spirit of SMPL these should still be considered expressions that return as their result, the value of the last expression evaluated. Note, to do this properly, you will also want to consider implementing the forms `break` and `continue`. **[10]**

Other optional features mentioned in the language specification will also attract extra credit if implemented. (Do not go overboard, there is a limit of 50% to extra credit, so your maximum score is 150/100). Regardless of the extensions that you choose to implement, you must ensure that your version is backwards compatible with the core specification of the language.

Use `JavaCUP` and `jflex` 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. You will probably want to continue with the approach used in Assignment 2, where a parent type was defined for all values, and it was bestowed with all the operations associated with syntactically recognised operators (such as `+, *, -`). If you recall, the parent class threw a type exception, so that in the event that a subtype did not support an operation, the exceptional behaviour was already implemented. If done properly, this can be a substantial amount of work, so it could also be an identified subtask for a group member. You can use the rest of the implementation of Assignment 2 as a guide for how to complete the rest of the interpretor.

Be sure that your interpreter can accept a sequence of files on the command line, and if the last file parameter is a "-" (or if no arguments are given at all), it should go into a REPL (Read-Eval-Print-Loop) to provide a kind of shell to the user. If files are provided, they should be evaluated in the order given. This way, if files are given, and then a REPL is entered, all of the definitions in the files will be available at the prompt of the REPL. Implementing your main class this way will save a lot of time during the checkoff. (The code that was given out for Assignment 2 already implements this behaviour, so if you reuse that class, you would already have taken care of this issue).

For checkoff, you should have ready to show me, any additional features that you may have implemented. You do not have to limit yourself to only extensions that I have described. It would be exciting to see some original language innovations coming from you guys. More than anything, I want you to have fun while implementing this interpreter, so try to make the language your own, in spite of the language specification.

Problem 2: *Programming in* SMPL **[10]**

Implement a minimising binary heap data structure in SMPL. It should support the operations `getMin`, `deleteMin`, and `insert` and all those operations should run in $O(\lg n)$ time (`getMin` should be in $O(1)$ time). Your implementation should also support a `heapify` function that will accept a vector, and return a binary heap containing all of the values in the input vector. Your implementation will be tested by subjecting it to insertions and extractions (deletions) and checking that the values returned from the `getMin` queries are appropriate given the values present at the time.