MCN 7105: STRUCTURE AND INTERPRETATION OF COMPUTER PROGRAMS

END OF SEMESTER PROJECT

Instructions

Deadline: 14th June 2021, 11PM. (Hard deadline: the deadline is FIXED and CANNOT be extended!)
Deliverables: Source code file and a report (max. 5 pages, PDF) as a single zip file.
Submission: Submit your solution via Muele. No email submissions please!
File Naming: Firstname-Lastname-project2.zip

The aim of this assignment project is to enable you understand how meta-linguistic abstractions can be used to cope with the complexity of software systems. The project provides an insight into how to build meta-linguistic abstractions using primitive operations and data provided by the underlying programming language. The assignment consists of an implementation and a brief report. The report should describe your system including your design decisions (many flaws of programming languages are as a result of poor design decisions). The report and submitted source code should clearly indicate a code snippet for each part of the assignment. The assignment will be graded and can be subjected to an additional defence. Finally, be sure that the submitted solution is your own!

Assignment

In Software Engineering as we confront increasingly complex problems, we will find that any fixed programming language is not sufficient for our needs and must therefore turn our attention to building new programming languages. In this assignment, you will build on the interpreters such as those presented in Chapter 4 of the textbook (source code provided on Muele m-eval.rkt).

Exercises

1. Load the code files into your Scheme environment. Notice how it now gives you a new prompt, in order to alert you to the fact that you are now "talking" to this new interpreter. Try evaluating some simple expressions in this interpreter. Note that this interpreter has no debugging mechanism, that is, if you hit an error, you will be thrown into the debugger for the underlying Scheme. This can be valuable in helping you to debug your code for the new interpreter, but you will need to restart the interpreter. You may quickly notice that some of the primitive procedures about which Scheme normally knows are missing in m-eval. These include some simple arithmetic procedures (such as *) and procedures such as cadr, cddr, newline. Extend your evaluator by adding these new primitive procedures (and any others that you think might be useful). Check through the code to figure out where this is done. In order to make these changes visible in the evaluator, you will need to rebuild the global environment.

2. Dynamic binding (or dynamic scoping) is an alternative design for procedures, in which the procedure body is evaluated in an environment obtained by extending the environment at the point of call. For example in the follow definitions:

```
( define x 1)
( define (f x) (g 2))
( define (g y) (+ x y))
(f 5) => 3 ; not 7
```

Here f and g are bound to procedures created in the initial environment. Because Scheme is statically (or lexically) scoped, the call to g from f extends the initial environment (the one in which g was created) with a binding of y to 2. In this extended environment, y is 2 and x is 1. (In a dynamically bound programming language, the call to g would extend the environment in effect during the call to f, in which x is bound to 5 by the call to f, and the answer would be 7). Modify the language to use dynamic binding.

3. The current version of our interpreter allows us to create new bindings for variables by means of define, but provides no way to get rid of bindings. Extend the interpreter with a special form unbind! that removes the binding of a given symbol from the environment in which the unbind! expression is evaluated. This problem is not completely specified. For example, should we remove only the binding in the first frame of the environment? Describe your specification and justify any choices you make. Implement your specification to work as follows:

```
> ( define y 10)
> y
>> 10
> ( unbind ! y)
> y
>> " Unbound variable y"
```

4. We have seen that our evaluator treats any compound expression as an application of a procedure, unless that expression is a special form that has been explicitly defined to obey a different set of rules of evaluation (e.g., define, lambda, if). Extend the evaluator to add do...while loop special form, common loop construct in other languages. It has the following syntax:

```
(do exp1
exp2
...
expN
while test )
```

The behavior is as follows: exp1 through expN are evaluated in sequence. Then test is evaluated. If test evaluates to #f then we stop executing the loop and return the symbol done; otherwise the loop is repeated from the start. For example,

```
(( let ((x '()))
(do (set! x ( cons '* x))
( display x)
( newline )
while (< ( length x) 3))) ; evaluate this to get
(*)
(* *)
(* * *)
; Value : done
```

THE END