COMP 335: Principles of Programming Languages

Assignment 2: Reference and Mutations

Due date: Tuesday October 28, 2014, at 11:59pm

Collaboration policy

- **This is a partner assignment.** Since we have an odd number of students, there can be some groups of three or singletons, but I have to approve of those groups before they start working.
- You may not share or look at each other's code outside of your group (short examples of Racket syntax are fine)
- You must give credit to any outside resources used (for example, non-course textbooks or wikipedia)
- Answers to written questions must be the work of your group only.

Handin instructions

One person from your group should upload the following files individually (no zip files, please) to the oncourse website:

- {<first initial><lastname>}+_hw2.pdf – PDF file with your answers to the written problems.
- {<first initial><lastname>}+_hw2.rkt – DrRacket definitions file with test expressions and code for the programming problem.
- {<first initial><lastname>}+_hw2_README.txt – (optional) Text file with anything else you wish to tell me.

Everyone else in the group should upload a README textfile with the names of your group members.

Overview: In class, we discussed one possible way to implement references to functions and identifiers in a programming language, by substituting each identifier in the source code with the value (or expression if we are using lazy evaluation) that it is bound to. However, there are some downsides to substitution:

1. The substitution has to traverse the entire source code of the program in order to substitute the values that are bound to the identifiers, and then traverse the code again in order to interpret the program. It would be nice to do everything in one step, particularly since there may be parts of the code (such as unvisited branches) that do not need to be visited at all.

2. While our Racket interpreter always has access to the source code, there are other languages where this is not the case. For example, in a compiled programming language, we might want to include code from a library which is already compiled, and we might not have access to the source code.

In this assignment, you will consider an *alternative* implementation strategy for implementing references which addresses these issues. Recall that the *environment* is all the bindings that are in scope at a particular point of the program. In this implementation, the interpreter will use a data structure to represent the environment while interpreting the program.

Programming (50 points)

You and your group will write up an implementation of the interpreter described in chapter 6 of the textbook. Almost all of the code needed is given word for word in the textbook, but you will need to figure out how it all fits together, and then write your own test expressions (with expected output!!) for the "interp" and "lookup" functions.

The semantic representation for expressions in the programming language, function definitions, and the parser will not change from the substitution-based implementation (which we covered in class the week of September 23, and is also in the textbook Chapter 5). For your convenience, I have provided these on the oncourse website (hw2_supportcode.rkt).

Grading:

- 10 points: Implement the environment data structure, the binding data-type, and the lookup function.
- 15 points: Write test expressions with expected output for the above pieces of the program. Some particular edge cases I want you to include are what happens if there are two bindings with the same identifier, and a test/exn case for if you try to lookup an identifier that isn't in the environment.
- 10 points: Implement the interp function.
- 15 points: Write test expressions with expected output for the interp function. You can include your test expressions from the previous assignment, but you will need to write new test cases that use function application. You may use the example function definitions in the support code or you may write your own. Make sure to write a test case that shows that you fixed the scoping problem described in section 6.3-6.4.

Writing 1 (10 points)

In this implementation, both the function definitions and the bindings in the environment are stored as list data structures. What is the worst case runtime (big-O) for the `get-fundef` and `lookup` functions? What kind of data structure could be used instead to improve the runtime? You may refer to your knowledge of data structures from other programming languages; you don't have to limit yourself to your knowledge of Racket or what you think the language seems to provide.

Writing 2 (10 points)

In the core semantic representation for the application of a function (the appC variant of ExprC), the argument to substitute for the identifier in body of the function has the *ExprC type*. However, the bindings are implemented as pairs of identifiers and values of the *number type*. What kind of evaluation strategy (lazy or eager) does this implementation of the interpreter use? Explain.

Writing 3 (30 points, 10 points each)

(Note: If your group would like to write the code for this you can do that if it helps you understand these questions, but do not hand it in. However, you should also try to understand intuitively what is happening.)

a. Right now your programming language only has two kinds of references: references to the function definitions, and references to the arguments inside the body of a function. Say that you want to add a pre-defined global variable for pi (rounded off as 3.1316) so that programmers can write expressions such as `(* 2 (* pi 15))` in your language. Describe **in English** all the places in the implementation you will need to change in order to add pi to your language.

b. Now that pi is a global variable in the language, let's say we also add the following function definition to the list myfds:

```
(fdC 'mypi 'pi (idC 'pi))
```

Now consider the three expressions below. Which ones will evaluate to the same number, and which ones will not?

```
(+ pi (mypi 5))
(+ (mypi 5) pi)
(mypi (+ pi 5))
```

c. Consider the following Python program (assuming you all know python).

```
pi = 3.1416

def mypi (pi):

   return pi + 5

print(mypi(5))

print(pi)
```

What is the output of the two print statements? How does this example this relate to the concepts of scope and immutiblility for environments?

Total: 100 points