CSCI-GA-2110 – Problem Set 2 – Written Part

This document describes the written exercises for problem set 2. Each exercise is designated as a "Task" in this document. Please write or type your solutions neatly to these tasks, produce a legible PDF clearly indicating where each question is answered, and upload the results to gradescope.

1 Encoding Let into Lambda

In class we discussed how a 'let' that binds a single variable can be defined in terms of a 'lambda' that takes a single argument. The idea was that:

```
(let ([x e1]) e2)
```

was equivalent to

```
(lambda (x) e2) e1
```

In the programming part of this assignment, you have to implement 'let' and 'let*' with multiple bindings. It turns out that these can also be encoded in terms of lambda.

Task 1.1 (4 pts). Demonstrate this encoding by showing how you would translate the following two examples into a version that uses single-argument lambdas instead of 'let'.

```
(let ([x e1] [y e2] [z e3]) e)
```

```
(let* ([x e1] [y e2] [z e3]) e)
```

2 Lexical Addresses

As we discussed in class, the particular names or symbols we use for variables do not matter in some sense: we can rename a variable at a binding site so long as we consistently rename all of the uses of that variable as well. In other words, the names of bound variables are just a way for us to refer to a particular "location" where a binding happened. One can make this idea prepcise by introducing the idea of *lexical address* of a variable. A lexical address consists of two components: a *lexical depth*, which tells us, which binding location the variable was bound in, and a *lexical position*, which tells us, of the multiple variables that may have been bound at that point, which one was this variable. To indicate the depth, we count (starting from 0) from the occurrence of the variable, how many binding sites we pass over in the abstract syntax tree to find where the variable is bound. For the position, we count (starting from 0) the location of the variable in that binding site.

Let's look at an example:

```
(lambda (x y)
  ((lambda (a)
     (a x y))
  x))
```

- On the third line, the depth of a is 0, because there are no intervening binding sites between where a was bound (the lambda on line 2) and this occurence of a. Its position is also 0, because it is the first variable bound by that lambda.
- On the third line, the depth of x is 1, because to get to the binding site for x, we pass over the enclosing site where a is bound. The position of x is 0, because it is the first variable bound at that site.
- On the third line, the depth of y is 1 (for the same reason as x's) but its position is 1, because it is the second argument in the lambda where it is bound.
- On the fourth line, the depth of x is 0, because to get to the binding site for x from that occurrence, there are no other binding sites that that x is in the scope of. Its position is also 0.

We can adopt a convention of annotating a variable with its depth and position, writing (x : d p) for when the variable x occurs with depth d and position p. Using this, the above program would become

```
(lambda (x y)
  ((lambda (a)
     ((a : 0 0) (x : 1 0) (y : 1 1)))
  (x : 0 0)))
```

Once we have annotated the variables with this information, we can see that the particular names of the variables are not necessary any longer to find where they are bound. In other words, we could just write (: d p) and we would know unambiguously which variable binding we meant. Similarly, we don't need to give the names of the arguments when binding them either, we just need to indicate how many arguments are bound. That is, we can rewrite the above as:

```
(lambda 2
  ((lambda 1
     ((: 0 0) (: 1 0) (: 1 1)))
  (: 0 0)))
```

where the 2 on line 1 indicates that that lambda had two arguments, while the 1 on line 2 indicates that that lambda had 1 argument.

We call this the *lexical address form* of the program.

Task 2.1 (3 pts). Convert the following Racket program into the lexical address form, similarly to the example above.

```
(lambda (x y z)
  (z (lambda (x y) (z x)) y))
```

Task 2.2 (3 pts). Consider the following program written in lexical address form:

```
(lambda 1
  (lambda 1
     (: 1 0)))
```

Write down an equivalent program written in standard Racket form with names for variables.