Week 2 Lab: Pattern Matching and Recursion

In this lab, we apply several of the programming language concepts we covered in lecture, including recursion, pattern matching and the idea of representing code using data structures like nested lists. This lab is an opportunity to practice writing functions using structural pattern-matching in a systematic way.

Task 1: Recursion and Tail Recursion (4 points)

In Racket, complete the function `replace-all` that replaces all instances of a symbol in a list with another symbol. Use pattern matching.

Then, rewrite this function so that it is tail recursive. In particular, complete the helper function `replace-all-helper`. (2 points)

Finally, write the same function again in Haskell: complete the function `replaceAll` that replaces all instances of a number in a list with another number. Write this function so that it is tail recursive. (2 points)

Task 2: Capture Avoiding Substitutions (6 points)

In this task, we will write a function that performs capture avoiding substitutions as described in lecture 1. This is task demonstrates that life is easier if you follow the structural approach to problem-solving.

Specifically, we will write a function `(subst expr id with)` that takes an expression `expr`, and substitutes occurrences of identifier `id` with the expression `with`. This function is the heart of a simple interpreter for the lambda calculus, and we will use this function in lecture 3.

Here's an example of how `subst` should behave.

```
> (subst '(f ((g a) (h a))) 'a '(b x))
'(f ((g (b x)) (h (b x))))
```

We get this result by replacing instances of the symbol `'a` with the expression `'(b x)` in our main expression `'(f ((g a) (h a)))`.

To be more precise, the function `subst` will take three arguments:

- The main expression. We will define valid expressions using a grammar below.
- An identifier to replace (a Racket symbol)
- An expression to replace the identifier with.

```
EXPRESSION GRAMMAR:

<expr> := SYMBOL
        | (lambda (SYMBOL) <expr>)
        | (<expr> <expr>)
```

The one caveat with the behaviour of `subst` is that in the function expression case `(lambda (SYMBOL) <expr>)`, if the lambda's parameter is the same as our identifier, then we will *not* modify the body expression. The reason is that in this situation, the parameter to the `lambda` **shadows** the outer variable with the same name. You may have seen other instances of variable shadowing in languages like Python, and the idea is the same.

Here are some examples:

```
> (subst '((lambda (a) (f a)) a) 'a 'b)
'((lambda (a) (f a)) b) ; since the lambda uses the parameter same name "a" as
                        ; the identifier we're trying to replace,
                        ; the body is left unchanged
> (subst '((lambda (x) (f a)) a) 'a 'b)
'((lambda (x) (f b)) b) ; if the lambda used a different parameter name, the body
                        ; would be updated
```

With that in mind, the remaining lab handout provides a step-by-step guide for implementing recursive functions like `subst`. Please reflect on the process carefully, so that you can apply similar processes in later labs and assignments.

Step 1

First, we should obtain a clear understanding of what the *input* to our function look like. This is to develop an understanding of what we need to implement, and as a stepping stone to writing some good test cases.

To that end, determine whether these are valid expressions in the grammar:

1. `a`
2. `b`
3. `(f a b)`
4. `((lambda (a) a) a)`
5. `((lambda (b) a) (lambda (a) a))`
6. `((lambda (a) (lambda (a) y)))`
7. `(b (lambda (a) (b b)))`

You may wish to generate some more valid (and interesting!) expressions using the above grammar.

Step 2

Now that we have some idea of what inputs would look like, let's develop an understanding of what the *outputs* should look like for each of these inputs.

For each of the valid expressions above, determine by hand what would happen if we call `(subst expr 'a 'c)` to replace instances of `'a` with `'c`.

You now have a collection of test cases that you can use to test your code as you go along. Implement these test cases in Racket. Order these test cases from simple to complex.

Step 3

We will follow the structural recursion approach to solving problems. In particular, there are three possible types of expressions, and they can be handled independently!

Start the function `subst`, and use pattern matching to match the three cases. In other words, this step involves writing only the **patterns** involved in the pattern matching.

Since pattern matching is checked top-to-bottom, we recommend matching the identifier case *last*.

```
(define/match (subst expr id with)
  [(expr id with) ; write pattern for lambda expressions
   expr           ; place holder for code to handle lambda expressions
  ]
  [(expr id with) ; write pattern for function calls
   expr           ; place holder for code to handle function calls
  ]
  [(expr id with) ; write simple pattern for identifiers: any expression that
                  ; didn't match the previous patterns should match here
   expr           ; place holder for code to handle function calls
  ])
```

Once again, we are only writing the **patterns** in this step, by replacing `(expr id with)` with the appropriate pattern.

We have structured the starter code so that its behaviour is identical to the identity function. This way, your function `subst` should be runnable every step of the way. Test your code to make sure that you did not introduce any syntax errors before moving to the next step.

Having runnable code every step of the way gives us a systematic way of making incremental progress. This may not be the most comfortable for you (yet!), but this is the way that experts write code. Practise writing code in this incrementally testable manner, and you will find it much easier (and less time-consuming!) to make progress on challenging problems.

Step 4

Start by handling the simplest of the three cases, namely the identifiers. This is your base case. How can you determine whether the identifier expression need to be replaced? What should you return in each case?

After this step, some of the tests you wrote in Step 2 should pass!

Step 5

The next case to handle is the function call case. This case is recursive. If you don't know how to start, go back to the expression grammar. The code for this case is actually deceptively short, but getting to the simple solution requires you to *trust the recursion*.

If you notice that you are reasoning about cases (e.g. what happens if the argument is a symbol? a lambda? Stop. You are currently writing a function that does this case analysis for you! **Trust the recursion**)

After this step, more of the tests you wrote in Step 2 should pass!

Step 6

Finally, complete the `lambda` case.

Submission and Instructions

Submit the files `w02lab.rkt` and `W02lab.hs` on Markus. To give you faster feedback on labs, you will be able to test your own code on Markus once every hour. However, it is up to you to debug and determine why your code pass or fail tests. On labs, the tests that you are able to run on Markus are the same tests that we will be running to determine your grade.

Please test your labs early. Markus sometimes hangs when running tests, particularly if you submit code that does not terminate. If so, you will not be able to continue to submit additional tests until your test times out.

For all labs and assignments:

- You may not import any libraries or modules unless explicitly told to do so.
- You may not use the function "eval" unless explicitly told otherwise.
- In Racket, you may not use any iterative or mutating functionality unless explicitly allowed. Iterative functions in Racket are functions like `loop` and `for`. Mutating functions in Racket have a `!` in their names, like `set!`. If use the materials discussed in class and do not go out of your way to find these functions, your code should be okay.
- The `(provide ...)` and `module (...) where` code in your files are very important. Please follow the instructions, and don't modify those lines of code! If you do so, your code will not be able to run and you will receive a grade of zero.
- Do not change the default Haskell language settings (i.e. by writing an additional line of code in the first line of your file)

Breaking any of these above rules can result in a grade of 0.

Moreover, code that cannot be imported (e.g., due to a syntax error, compilation error, or runtime error during import) will receive a grade of zero. Please make sure to run and test all of your code before your final submission.

You are permitted (and encouraged!) to write helper functions freely.