Lambda_Functions_S11

Lambda Functions

Lambda functions are actually the fundamental way in which Scheme defines a function:

```
(define (myFunc x y)
   (...))
```

is really just a Scheme shorthand for

```
(define myFunc (lambda (x y)
   (...)))
```

The only reason we ever give associate a name with something is if we wish to reference it more than once!

This means that everything that we done with functions already also applies to lambdas, since they are all the same thing.

The lambda function definition is useful when we want to define a function right where we use it. (Most likely because we only need to use that function once.)

For instance, lambdas are very useful when a function is needed as an input parameter, such as when calling higher-order functions, e.g. filter, map, foldr, etc.:

```
(filter (lambda (x) (...)) aList)

(map (lambda (x) (...)) aList)

(foldr (lambda (aFirst aRecursiveResult)(...)) aList)
```

Lambdas are also useful when a function is needed for a return value:

```
;; make-addX: num -> (lambda num -> num)
;; returns a function that adds its input to the original x input.
;; Examples:
(check-expect ((make-addX 3) 5) 8)
(check-expect ((make-addX 3) -2) 1)
(check-expect ((make-addX -5) 5) 0)
(check-expect ((make-addX -5) -2) -7)
(define (make-addX x)
   (lambda (y) (+ x y)))
```

Problem: We can't write a recursive algorithm using a lambda function because the lambda cannot refer to itself! This issue can be solved using more sophisticated lambda function techniques, which are mostly out of the scope of this course, and in Java, which always defines a name for object to use to reference itself.

Currying

Not a reference to cooking up the delicious spicy South Asian dish, but a rather to the American mathematician, Haskell Curry, for which the computer language "Haskell" is named. (To note, please see the link above for currying to see that Haskell Curry was not the original inventor of the process that now bears his name.)

Currying is technically the process in which a function, which takes n input parameters, creates another function, which only takes n- input parameters. Mathamatically, this is a type of process is called a "transformation". We can see the process in the "make_addX" example above, where the make-addX function is a function of 2 input parameters when we consider that in order to get a final result that is a number, we must supply 2 input parameters as such: ((make-addX num1) num2). See the "check-expect" tests for examples of this. However, the return value of the make-addX function is a function of only one input parameter. To get a numerical result then, we only need to supply 1 input parameter:

```
;; two input parameters needed to make numerical result:
((make-addX 5) 3)  ;; --> 8

;; but looking just at the return value of make-addX
(define add5 (make-addX 5))

;; only one input parameter is needed:
(add5 3) ;; --> 8
```

What does this mean? The effect is that the n'th input parameter in the original function, which by its very nature of being an input parameter, is a **variant** entity with regards to the usage of that function, "make-addX" above, but has been tranformed into an **invariant** entity with respect to the usage of the returned function, "add5" above.

```
(define add7 (make-addX 7)) ;; 7 is the value of the variant parameter "x" for make-addX
(define add42 (make-addX 42)) ;; 42 is the value of the variant parameter "x" for make-addX

(add7 10) ;; --> 17 because 7 is invariant now for add7
(add7 -15) ;; --> -8 because 7 is invariant now for add7

(add42 10) ;; --> 52 because 42 is invariant now for add42
(add42 -15) ;; --> 27 because 42 is invariant now for add42
```

One way to look at the currying process is to say that the returned lambda of make-addX "captures" the value of "x" *at the moment that it is created*. This capturing process will become very important later on when we have more complicated algorithms where we will want to clearly delineate parameters that are variant in one part of the code but invariant in another. The use of lambdas and their ability to curry variables will become one of our most powerful tools as the semester progresses!

Another way to think of currying is that it creates a scoping that lasts for longer than the lifetime of the function in which it is defined. That is, the returned function from make_addX retains the scoping defined by the running of make_addX long after make_addX has finished executing. All of this is part of the notion of a function's "closure", a topic we will discuss more formally very soon,

In the language of Design Patterns, make-addX is called a "factory" because it "manufactures" a function that we can use later.

Some important consequences of currying that we will exploit:

- Using invariant code, we can generate variant functions that behave differently simply by supplying different input values to the factory function that made them.
- The functions made by the factory can be used elsewhere in our system, meaning that we can dynamically change one part of the system and have those changes be reflected as different behaviors in a different part of the system, even though the code does not change at all. In effect, we can have parts of our system interact with each other without directly coupling their code together. This will become hugely important when we try to build larger, more complex systems.