CS403: Programming Languages

Functions

Key ideas

- first class status of functions
- passing functions
- returning functions
- returned functions ”remember” their free variables
- local variables via lambda

Functions as first-class objects

Suppose you really, really, really, hated the prefix notation of Scheme. The fact that functions are first-class objects gives you a way to force prefix notation into (nearly) infix. Suppose we wanted to add the integers 3 and 5 together. In Scheme, the answer would be the value of the expression:

```
(+ 3 5)
```

Unfortunately, there’s no way to have the Scheme interpreter make sense of the expression:

```
(3 + 5)
```

since it will attempt the execute the function 3. But we can come close by defining a function which does understand infix notation. Suppose we name our new function *expr*. Then we could call *expr* with an infix notation, as in:

```
(expr 3 + 5)
```

which we assume would evaluate to 8. Let’s define it now:

```
(define (expr left op right)
    (op left right)
    )
```

The first-class status of functions in Scheme make this transformation trivial. Extending *expr* to handle an arbitrary infix expression is not trivial, although not because of Scheme itself but because of the intricacies of associativity, precedence, and grouping of infix mathematical expressions. Suppose we say that all operations have the same precedence and that operators are evaluated from left to right and there is no grouping of sub-expressions. Then *expr* becomes:

```
(define (expr left op right . more)
    (cond
        ((null? more) (op left right))
        (else (apply expr (cons (op left right) more)))
        )
    )
```

Evaluating the expression:

```
(expr 3 + 4 - 5 * 2)
```

yields 4, the correct answer (remember there is no precedence).
This version of *expr* uses the syntax for supplying a variable number of argument (the ”dot” in the formal parameter list - extraneous arguments are bundled up in to the list more) and a number of functions for manipulating lists (*null?*, *apply*, and *cons* - more about these later). The point here is not so much understanding how this version of `expr` works but that it is quite powerful yet only slightly more complicated than the original version.
Adding associativity, precedence, and grouping requires the ability to “parse” expressions, which is the subject of the latter half of this course.

Functions which return functions

Previously, we've seen examples of functions which take other functions as parameters. Now let's look at functions which return functions. Consider this definition:

```
(define (adjuster op amount)
    (define (f x)
        (op x amount)
        )
    f
    )
```

This function defines an internal function, bound to the name *f*, and then returns the value of *f*, namely the internal representation of the function itself. Clearly, this function returned takes a single argument and then adjusts that argument based upon the operator *op* and the amount passed to the creating function. What happens when these definitions and expressions are evaluated?

```
(define adjust (adjuster + 2))
(adjust 10)
```

The name *adjust* is bound to a function, created by *adjuster*, which takes a single argument and "adjusts" it by adding 2 its argument. Therefore, evaluation of `(adjust 10)` yields 12. More than one kind of adjuster function can be created:

```
(define adjust1 (adjuster + 2))
(define adjust2 (adjuster * 3))
(adjust1 (adjust2 10))
```

Evaluation of the last expression should yield 32. Note that *adjust1* is unaffected by the creation of *adjust2*. They each "remember" their own adjustment operator and adjustment amount. Note that *op* and *amount* are free variables with respect to the returned function. In general, returned functions "remember" their free variables (by storing a pointer to those variables).

Function literals

It seems kind of klunky for the adjuster function to have to name the function it creates since nobody uses that name. It's similar to being required to write:

```
double pi()
    {
    return 3.14159;
    }
```

as:

```
double pi()
    {
    double x = 3.14159;
    return x;
    }
```

in C or C++. Why do we have to name a function object just to return its value? It turns out that in Scheme, you don't. Since functions are first class objects, they also have literal forms (above, *x* is a double variable and 3.14159 is a double literal). Let's adjust adjuster so that it simply returns the literal form of the function it creates.

```
(define (adjuster op amount)
    (lambda (x) (op x amount))
    )
```

Note that the literal form is very similar to the original definition. Only two changes have been made. First the name define is changed to *lambda* and the defined function name is deleted (since it's not needed anymore). Now let's look again at the use of adjuster. Note that the last three expressions are all equivalent and are generated by successive substitutions.

```
(define adjust (adjuster + 2))
(adjust 10)
((adjuster + 2) 10)
((lambda (x) (+ 2 x)) 10)
```

Using lambdas to generate local variables

One surprising consequence of first-class functions is that it is never necessary to have local variables in your language. Consider the function:

```
    f(x) = (x-1) * (x-1) - (x-1)
```

If we implement this function as a Scheme procedure, we get:

```
    (define (f x)
        (- (* (- x 1) (- x 1)) (- x 1))
        )
```

Note that the quantity `(- x 1)` is computed three times. It would be nice to be able to precompute this quantity, bind it to a name and then perform the calculation on the bound name, saving two recomputations of the quantity. If we don't want to define local variables, We can get by with a helper function:

```
(define (f x)
    (define (helper a)
        (- (* a a) a)
        )
    (helper (- x 1))
    )
```

Note how the quantity $x - 1$ is computed only once.
Still it seems awkward to have to name a function just to invoke it. Here's where *lambda*s come in. We can rewrite this function to use a call to a *lambda* which takes one argument and then pass the quantity `(- x 1)` as that argument.

```
    (define (f x)
        ((lambda (a) (- (* a a) a))
            (- x 1))
        )
```

I've placed the *lambda* expression on one line and its argument on the next line (indented) to help comprehension. Now the value of `(f x)` is the value of passing `(- x 1)` to the function (with no name) which subtracts its argument from the square of its argument. As the text says, this is such a useful way to construct local variables that a rearranged version of *lambda*, called *let*, is provided. Recast using *let*, the function becomes

```
    (define (f x)
        (let
            ((a (- x 1)))
            (- (* a a) a)
            )
        )
```

Even so, I find this a bit hard to read. So I use local variables in most situations:

```
    (define (f x)
        (define a (- x 1))
        (- (* a a) a)
        )
```

Care must be taken with internal defines in Scheme as they are as processed simultaneously. For example:

```
    (define a 16)

    (define (f x)
        (define b (* x a)) ; which a, local or global?
        (define a 3)
        (- (* a a) b)
        )
```

might not behave as expected.