Programming Languages

The Lambda Calculus and Church Numerals

Computability

It is thought that a Turing machine is capable of computing anything that is computable. An alternative to Turing machines, derived at roughly the same time, was Church's *lambda calculus*. The lambda calculus is easy to describe syntactically:

```
expression : IDENTIFIER
           | (lambda (IDENTIFIER) expression)
           | (expression expression)
```

It is also easy to describe semantically. The first alternative form of an expression, `IDENTIFIER`, is just a symbol. The second alternative corresponds to a function definition, while the third alternative corresponds to a function call. These simple rules describe a programming language that is thought to be capable, like a Turing machine, of computing anything that is computable.

From the calculus, we can see that functions can only take one argument and that the body of a function is composed of a single expression. A striking feature of the calculus is the lack of numbers. Numbers seem rather critical to many computations, but have no fear. Church came up with a clever way to represent numbers (integers) using the lambda calculus.

Church numerals

Here are the first three Church numerals, that represent the integers *zero*, *one*, and *two*:

```
(lambda (f) (lambda (x) x))
(lambda (f) (lambda (x) (f x)))
(lambda (f) (lambda (x) (f (f x))))
```

We can see that these numerals take two arguments, f and x, with the first one f being curried. The argument f is a function, while the argument x is a suitable argument to f itself. We can also see that *zero*, the first expression, calls the function f zero times, while *one* calls f once and *two* calls it twice, feeding the result of the first call into the second. By extrapolation, we might guess that the Church numeral *three* would be equivalent to the expression:

```
(lambda (f) (lambda (x) (f (f (f x)))))
```

and we would be right. If we think of x as some base value and f as an *incrementing* function, then we can interpret the Church numeral *three* as incrementing some base value three times. We call the successive applications of the incrementor to the base value a *rendering*. We can actually see a rendering of a Church numeral in action. Suppose we define both *three* and an incrementing function in Scheme:

```
(define three (lambda (f) (lambda (x) (f (f (f x))))))
(define (inc z) (+ z 1))
```

Using an integer 0 as our base value, we can force a rendering of *three* to show that it is indeed a representation of the integer 3. This expression:

```
(inspect ((three inc) 0))
```

prints out:

```
((three inc) 0) is 3
```

By calling any Church numeral (it is a function!) with an incrementor and a base value, we force a rendering.

Manipulating Church numerals

Using the Scheme, we can define lambda calculus-like mathematical functions that manipulate Church numerals. Here is the successor function:

```
(define succ
    (lambda (n)
        (lambda (f)
            (lambda (x)
                (f ((n f) x))          ; ((n f) x) is a rendering of n
                )
            )
        )
    )
```

We can test our successor function by using the above-defined incrementor function and a base value of integer 0. The expression:

```
    (inspect (((succ three) inc) 0))
```

prints out:

```
    (((succ three) inc) 0) is 4
```

as expected. The successor function works because we apply the incrementor one more time to the rendering of the original number.

We can also add two Church numerals:

```
    (define add
        (lambda (a)
            (lambda (b)
                (lambda (f)
                    (lambda (x)
                        ((a f) ((b f) x))    ; ((b f) x) is a rendering of b
                        )
                    )
                )
            )
        )
```

Here, we render the addend b and use that rendering as the base value for the augend a. Thus, in a rendering of the sum, the original base value gets incremented b times and then is further incremented a times, for a total of $a + b$ increments over the base value.

Church numerals repeatedly apply a function to a base value. We can use this fact to run *any* single-argument function repeatedly on *any* value. This is what a multiplication function does:

```
    (define mul
        (lambda (a)
            (lambda (b)
                (lambda (f)
                    (lambda (x)
                        ((((a (add b)) zero) f) x)     ; ((a (add b)) zero) renders a Church numeral
                        )
                    )
                )
            )
        )
```

Here, we render a with a curried version of *add* on the Church numeral *zero*. The *add* function normally takes two arguments, but by currying the first argument, we turn in into a one argument function. If a was the Church numeral *one*, we would apply the curried *add* function once to *zero*, yielding:

```
    ((add b) zero)
```

which would give us just b. If a was *two*, we would apply the curried *add* function twice:

```
    ((add b) ((add b) zero))
```

If a were *three*, we'd get:

```
    ((add b) ((add b) ((add b) zero)))
```

Notice in each of these cases, we add in b, starting with *zero*, a number of times, yielding the expected product. Once all the repeated addition is completed, we render the resulting Church numeral with the incrementor f and base value x.