CSCI.4430/6969 Programming Languages
Lecture Notes

August 26, 2003

The mathematical notation for defining a *function* is with a statement such as

$$f(x) = x^2, \quad f : \mathbf{Z} \to \mathbf{Z},$$

where $\mathbf{Z}$ is the set of all integers. The first $\mathbf{Z}$ is called the *domain* of the function, or the set of values x can take. The second $\mathbf{Z}$ is called the *range* of the function, or the set containing all possible values of $f(x)$.

Suppose $f(x) = x^2$ and $g(x) = x + 1$. Traditional function *composition* is defined as

$$f \circ g = f\left(g(x)\right).$$

With our functions f and g,

$$f \circ g = f\left(g(x)\right) = f(x+1) = x^2 + 2x + 1.$$

Similarly

$$g \circ f = g\left(f(x)\right) = g(x^2) = x^2 + 1.$$

Therefore, function composition is not commutative.

In lambda (λ) calculus, we can use a different notation to define these same concepts. To define a function $f(x) = x^2$, we instead write

$$\lambda x.x^2.$$

Similarly for $g(x) = x + 1$ we instead write

$$\lambda x.x + 1.$$

To describe a function *application* such as $f(2) = 4$, we write

$$(\lambda x.x^2 \quad 2) \Rightarrow 2^2 \Rightarrow 4.$$

The *syntax* for lambda calculus expressions is

$$\begin{array}{rlll} e ::= & v & & \text{– variable} \\ | & \lambda v.e & & \text{– lambda expression} \\ | & (e & e) & \text{– procedure call} \end{array}$$

The *semantics* of the lambda calculus, or the way of evaluating or simplifying expressions, is defined by the rule

$$(\lambda x.E \quad M) \Rightarrow E\{M/x\}.$$

The new expression $E\{M/x\}$ can be read as "replace 'fresh' x's in E with M". Informally, a "fresh" x is an x that is not nested inside another lambda expression. We will cover free and bound variable occurrences in detail in upcoming lectures.

For example, in the expression

$$(\lambda x.x^2 \quad 2),$$

$E = x^2$ and $M = 2$. To evaluate the expression, we replace x's in E with M, to obtain

$$(\lambda x.x^2 \quad 2) \Rightarrow 2^2 \Rightarrow 4.$$

In lambda calculus, all functions may only have one variable. Functions with more than one variable may be expressed as a function of one variable through *currying*. Suppose we have a function of two variables expressed in the normal way

$$h(x, y) = x + y, \quad h : (\mathbf{Z} \times \mathbf{Z}) \rightarrow \mathbf{Z}.$$

With currying, we can input one variable at a time into separate functions. The first function will take the first argument, x, and return a function that will take the second variable, y, and will in turn provide the desired output. To create the same function with currying, let

$$f : \mathbf{Z} \rightarrow (\mathbf{Z} \rightarrow \mathbf{Z})$$

and

$$g : \mathbf{Z} \rightarrow \mathbf{Z}.$$

That is, f maps integers to a function, and g maps integers to integers. The function $f(x)$ returns a function g_x that provides the appropriate result when supplied with y. For example,

$$f(2) = g_2, \quad \text{where} \quad g_2(y) = 2 + y.$$

So

$$f(2)(3) = g_2(3) = 2 + 3 = 5.$$

In lambda calculus this function would be described with currying by

$$\lambda x.\lambda y.x + y.$$

For function application, we nest two application expressions

$$((\lambda x.\lambda y.x + y \quad 2) \quad 3).$$

We may then simplify this expression using the semantic rule (also called beta (β) reduction)

$$((\lambda x.\lambda y.x + y \quad 2) \quad 3) \Rightarrow (\lambda y.2 + y \quad 3) \Rightarrow 2 + 3 \Rightarrow 5.$$

The composition operation $\circ$ can itself be considered a function (also called *higher-order* function) that takes two other functions as its input and returns a function as its output; that is if the first function is $\mathbf{Z} \to \mathbf{Z}$ and the second function is also $\mathbf{Z} \to \mathbf{Z}$, then

$$\circ : (\mathbf{Z} \to \mathbf{Z}) \times (\mathbf{Z} \to \mathbf{Z}) \to (\mathbf{Z} \to \mathbf{Z}).$$

We can also define function composition in lambda calculus. Suppose we want to compose the square function and the increment function, defined as

$$\lambda x.x^2 \quad \text{and} \quad \lambda x.x + 1.$$

We can define function composition as a function itself with currying by

$$\lambda f.\lambda g.\lambda x.(f \quad (g \quad x)).$$

Applying two variables to the composition function with currying works the same way as before, except now our variables are functions.

$$\begin{array}{rl} & ((\lambda f.\lambda g.\lambda x.(f \quad (g \quad x)) \quad \lambda x.x^2) \quad \lambda x.x + 1) \\ \Rightarrow & (\lambda g.\lambda x.(\lambda x.x^2 \quad (g \quad x)) \quad \lambda x.x + 1) \\ \Rightarrow & \lambda x.(\lambda x.x^2 \quad (\lambda x.x + 1 \quad x)). \end{array}$$

The resulting function gives the same results as $f(g(x)) = (x + 1)^2$.

In the Scheme programming language we can use lambda calculus expressions. They are defined using a similar syntax. To define a function we use the code

```
(lambda(x [y z ...])  expr)
```

where additional variables `y`, `z`, etc. are optional. Scheme syntax allows you to have functions of more than one variable. The code

```
(lambda(x) (* x x))
```

describes the square function. Note that even common operations are considered functions and are always used in a prefix format. You may define variables (which may themselves be functions) with

```
(define a b).
```

For example,

```
(define f (lambda(x) (* x x)))
```

defines a function $f(x) = x^2$. To perform a procedure call, use the code

```
(f x [y z ...])
```

where `y`, `z`, etc. are additional parameters that `f` may require. The code

```
(f 2)
```

evaluates $f(2) = 4$.