Functional Languages

11.7 Theoretical Foundations

EXAMPLE 11.77
Functions as mappings

Mathematically, a function is a single-valued mapping: it associates every element in one set (the *domain*) with (at most) one element in another set (the *range*). In conventional notation, we indicate the domain and range by writing

$$\text{sqrt} : \mathcal{R} \longrightarrow \mathcal{R}$$

We can, of course, have functions of more than one variable—that is, functions whose domains are Cartesian products:

$$\text{plus} : [\mathcal{R} \times \mathcal{R}] \longrightarrow \mathcal{R}$$

If a function provides a mapping for every element of the domain, the function is said to be *total*. Otherwise, it is said to be *partial*. Our *sqrt* function is partial: it does not provide a mapping for negative numbers. We could change our definition to make the domain of the function the non-negative numbers, but such changes are often inconvenient, or even impossible: inconvenient because we should like all mathematical functions to operate on $\mathcal{R}$; impossible because we may not know which elements of the domain have mappings and which do not. Consider for example the function f that maps every natural number a to the smallest natural number b such that the digits of the decimal representation of a appear b digits to the right of the decimal point in the decimal expansion of π. Clearly $f(59) = 4$, because $\pi = 3.14159\ldots$. But what about $f(428945028)$, or in general $f(n)$ for arbitrary n? Absent results from number theory, it is not at all clear how to characterize the values at which f is defined. In such a case a partial function is essential.

EXAMPLE 11.78
Functions as sets

It is often useful to characterize functions as sets or, more precisely, as subsets of the Cartesian product of the domain and the range:

$$\text{sqrt} \subset [\mathcal{R} \times \mathcal{R}]$$
$$\text{plus} \subset [\mathcal{R} \times \mathcal{R} \times \mathcal{R}]$$

We can specify *which* subset using traditional set notation:

$$\text{sqrt} \equiv \left\{(x, y) \in \mathcal{R} \times \mathcal{R} \mid y > 0 \wedge x = y^2\right\}$$
$$\text{plus} \equiv \{(x, y, z) \in \mathcal{R} \times \mathcal{R} \times \mathcal{R} \mid z = x + y\}$$

Note that this sort of definition tells us what the value of a function like sqrt is, but it does *not* tell us how to compute it; more on this distinction below.

EXAMPLE 11.79
Functions as powerset elements

One of the nice things about the set-based characterization is that it makes it clear that a function is an ordinary mathematical object. We know that a function from A to B is a subset of $A \times B$. This means that it is an *element* of the *powerset* of $A \times B$—the set of all subsets of $A \times B$, denoted $2^{A \times B}$:

$$\text{sqrt} \in 2^{\mathcal{R} \times \mathcal{R}}$$

Similarly,

$$\text{plus} \in 2^{\mathcal{R} \times \mathcal{R} \times \mathcal{R}}$$

Note the overloading of notation here. The powerset 2^A should not be confused with exponentiation, though it is true that for a finite set A the number of elements in the powerset of A is 2^n, where $n = |A|$, the cardinality of A.

EXAMPLE 11.80
Function spaces

Because functions are single-valued, we know that they constitute only *some* of the elements of $2^{A \times B}$. Specifically, they constitute all and only those sets of pairs in which the first component of each pair is unique. We call the set of such sets the *function space* of A into B, denoted $A \to B$. Note that $(A \to B) \subset 2^{A \times B}$. In our examples:

$$\text{sqrt} \in [\mathcal{R} \to \mathcal{R}]$$
$$\text{plus} \in [(\mathcal{R} \times \mathcal{R}) \to \mathcal{R}]$$

EXAMPLE 11.81
Higher-order functions as sets

Now that functions are elements of sets, we can easily build higher-order functions:

$$\text{compose} \equiv \{(f, g, h) \mid \forall x \in \mathcal{R},\ h(x) = f(g(x))\}$$

What are the domain and range of compose? We know that f, g, and h are elements of $\mathcal{R} \to \mathcal{R}$. Thus

$$\text{compose} \in [(\mathcal{R} \to \mathcal{R}) \times (\mathcal{R} \to \mathcal{R})] \to (\mathcal{R} \to \mathcal{R})$$

Note the similarity to the notation employed by the ML type system (Section 7.4).

EXAMPLE 11.82
Curried functions as sets

Using the notion of "currying" from Section 11.6, we note that there is an alternative characterization for functions like plus. Rather than a function from pairs of reals to reals, we can capture it as a function from reals to functions from reals to reals:

$$\text{curried\_plus} \in \mathcal{R} \to (\mathcal{R} \to \mathcal{R})$$

We shall have more to say about currying in Section C-11.7.3.

11.7.1 Lambda Calculus

As we suggested in the main text, one of the limitations of the function-as-set notation is that it is *nonconstructive*: it doesn't tell us how to *compute* the value of a function at a given point (i.e., on a given input). Church designed the lambda calculus to address this limitation. In its pure form, lambda calculus represents *everything* as a function. The natural numbers, for example, can be represented by a distinguished zero function (commonly the identity function) and a successor function. (One common formulation uses a select_second function that takes two arguments and returns the second of them. The successor function is then defined in such a way that the number n ends up being represented by a function that, when applied to select_second n times, returns the identity function [Mic89, Sec. 3.5]; [Sta95, Sec. 7.6]; see Exercise C-11.23.) While of theoretical importance, this formulation of arithmetic is highly cumbersome. We will therefore take ordinary arithmetic as a given in the remainder of this subsection. (And of course all practical functional programming languages provide built-in support for both integer and floating-point arithmetic.)

A lambda expression can be defined recursively as (1) a *name*; (2) a lambda *abstraction* consisting of the letter λ, a name, a dot, and a lambda expression; (3) a function *application* consisting of two adjacent lambda expressions; or (4) a parenthesized lambda expression. To accommodate arithmetic, we will extend this definition to allow numeric literals.

EXAMPLE 11.83
Juxtaposition as function application

When two expressions appear adjacent to one another, the first is interpreted as a function to be applied to the second:

sqrt n

Most authors assume that application associates left-to-right (so $f\ A\ B$ is interpreted as $(f\ A)\ B$, rather than $f\ (A\ B)$), and that application has higher precedence than abstraction (so $\lambda x.A\ B$ is interpreted as $\lambda x.(A\ B)$, rather than $(\lambda x.A)\ B$). ML adopts these rules.

EXAMPLE 11.84
Lambda calculus syntax

Parentheses are used as necessary to override default groupings. Specifically, if we distinguish between lambda expressions that are used as functions and those that are used as arguments, then the following unambiguous CFG can be used to generate lambda expressions with a minimal number of parentheses:

expr $\longrightarrow$ `name` | *number* | λ `name` `.` *expr* | *func arg*
func $\longrightarrow$ `name` | `(` λ `name` `.` *expr* `)` | *func arg*
arg $\longrightarrow$ `name` | *number* | `(` λ `name` `.` *expr* `)` | `(` *func arg* `)`

In words: we use parentheses to surround an abstraction that is used as either a function or an argument, and around an application that is used as an argument.

EXAMPLE 11.85
Binding parameters with λ

The letter λ introduces the lambda calculus equivalent of a formal parameter. The following lambda expression denotes a function that returns the square of its argument:

$$\lambda x.\text{times } x\ x$$

The name (variable) introduced by a λ is said to be *bound* within the expression following the dot. In programming language terms, this expression is the variable's scope. A variable that is not bound is said to be *free*.

EXAMPLE 11.86
Free variables

As in a lexically scoped programming language, a free variable needs to be defined in some surrounding scope. Consider, for example, the expression $\lambda x.\lambda y.\text{times } x\ y$. In the inner expression ($\lambda y.\text{times } x\ y$), y is bound but x is free. There are no restrictions on the use of a bound variable: it can play the role of a function, an argument, or both. Higher-order functions are therefore completely natural.

EXAMPLE 11.87
Naming functions for future reference

If we wish to refer to them later, we can give expressions names:

$$\begin{aligned} \text{square} &\equiv \lambda x.\text{times } x\ x \\ \text{identity} &\equiv \lambda x.x \\ \text{const7} &\equiv \lambda x.7 \\ \text{hypot} &\equiv \lambda x.\lambda y.\text{sqrt (plus (square } x\text{) (square } y\text{))} \end{aligned}$$

Here $\equiv$ is a metasymbol meaning, roughly, "is an abbreviation for."

EXAMPLE 11.88
Evaluation rules

To compute with the lambda calculus, we need rules to evaluate expressions. It turns out that three rules suffice:

beta reduction: For any lambda abstraction $\lambda x.E$ and any expression M, we say

$$(\lambda x.E)\ M \rightarrow_\beta E[M \backslash x]$$

where $E[M \backslash x]$ denotes the expression E with all free occurrences of x replaced by M. Beta reduction is not permitted if any free variables in M would become bound in $E[M \backslash x]$.

alpha conversion: For any lambda abstraction $\lambda x.E$ and any variable y that has no free occurrences in E, we say

$$\lambda x.E \rightarrow_\alpha \lambda y.E[y \backslash x]$$

eta reduction: A rule to eliminate "surplus" lambda abstractions. For any lambda abstraction $\lambda x.E$, where E is of the form $F\ x$, and x has no free occurrences in F, we say

$$\lambda x.F\ x \rightarrow_\eta F$$

EXAMPLE 11.89
Delta reduction for arithmetic

To accommodate arithmetic we will also allow an expression of the form op x y, where x and y are numeric literals and op is one of a small set of standard functions, to be replaced by its arithmetic value. This replacement is called *delta reduction*. In our examples we will need only the functions plus, minus, and times:

$$
\begin{array}{lll}
 & (\underline{\lambda f}.\lambda g.\lambda h.f\,g\,(h\,h))\underline{(\lambda x.\lambda y.x)}h(\lambda x.x\,x) & \\
\rightarrow_\beta & (\lambda g.\underline{\lambda h}.(\lambda x.\lambda y.x)g(\underline{h\,h}))h(\lambda x.x\,x) & (1)\\
\rightarrow_\alpha & (\underline{\lambda g}.\lambda k.(\lambda x.\lambda y.x)g(k\,k))\underline{h}(\lambda x.x\,x) & (2)\\
\rightarrow_\beta & (\underline{\lambda k}.(\lambda x.\lambda y.x)h(k\,k))\underline{(\lambda x.x\,x)} & (3)\\
\rightarrow_\beta & (\underline{\lambda x}.\lambda y.x)\underline{h}((\lambda x.x\,x)\,(\lambda x.x\,x)) & (4)\\
\rightarrow_\beta & (\underline{\lambda y}.h)\underline{((\lambda x.x\,x)\,(\lambda x.x\,x))} & (5)\\
\rightarrow_\beta & h & (6)
\end{array}
$$

Figure 11.5 **Reduction of a lambda expression.** The top line consists of a function applied to three arguments. The first argument (underlined) is the "select first" function, which takes two arguments and returns the first. The second argument is the symbol h, which must be either a constant or a variable bound in some enclosing scope (not shown). The third argument is an "apply to self" function that takes one argument and applies it to itself. The particular series of reductions shown occurs in normal order. It terminates with a simplest (normal) form of simply h.

$$
\begin{array}{rrl}
\text{plus } 2\,3 & \rightarrow_\delta & 5\\
\text{minus } 5\,2 & \rightarrow_\delta & 3\\
\text{times } 2\,3 & \rightarrow_\delta & 6
\end{array}
$$

Beta reduction resembles the use of call by name parameters (Section 9.3.1). Unlike Algol 60, however, the lambda calculus provides no way for an argument to carry its referencing environment with it; hence the requirement that an argument not move a variable into a scope in which its name has a different meaning. Alpha conversion serves to change names to make beta reduction possible. Eta reduction is comparatively less important. If square is defined as above, eta reduction allows us to say that

EXAMPLE 11.90
Eta reduction

$$\lambda x.\text{square } x \rightarrow_\eta \text{square}$$

In English, square is a function that squares its argument; $\lambda x.$square x is a function of x that squares x. The latter reminds us explicitly that it's a function (i.e., that it takes an argument), but the former is a little less messy looking.

Through repeated application of beta reduction and alpha conversion (and possibly eta reduction), we can attempt to reduce a lambda expression to its simplest possible form—a form in which no further beta reductions are possible. An example can be found in Figure C-11.5. In line (2) of this derivation we have to employ an alpha conversion because the argument that we need to substitute for g contains a free variable (h) that is bound within g's scope. If we were to make the substitution of line (3) without first having renamed the bound h (as k), then the free h would have been *captured*, erroneously changing the meaning of the expression.

EXAMPLE 11.91
Reduction to simplest form

In line (5) of the derivation, we had a choice as to which subexpression to reduce. At that point the expression as a whole consisted of a function application in which the argument was itself a function application. We chose to substitute the main argument $((\lambda x.x\ x)\ (\lambda x.x\ x))$, unevaluated, into the body of the main lambda abstraction. This choice is known as *normal-order* reduction, and corresponds to normal-order evaluation of arguments in programming languages, as discussed in Sections 6.6.2 and 11.5. In general, whenever more than one beta reduction could be made, normal order chooses the one whose λ is left-most in the overall expression. This strategy substitutes arguments into functions before reducing them. The principal alternative, *applicative-order* reduction, reduces both the function part and the argument part of every function application to the simplest possible form before substituting the latter into the former.

Church and Rosser showed in 1936 that simplest forms are unique: any series of reductions that terminates in a nonreducible expression will produce the same result. Not all reductions terminate, however. In particular, there are expressions for which no series of reductions will terminate, and there are others in which normal-order reduction will terminate but applicative-order reduction will not.

EXAMPLE 11.92
Nonterminating applicative-order reduction

The example expression of Figure C-11.5 leads to an infinite "computation" under applicative-order reduction. To see this, consider the expression at line (5). This line consists of the constant function $(\lambda y.h)$ applied to the argument $(\lambda x.x\ x)\ (\lambda x.x\ x)$. If we attempt to evaluate the argument before substituting it into the function, we run through the following steps:

$$\begin{aligned} & (\lambda x.x\ x)\ \underline{(\lambda x.x\ x)} \\ \rightarrow_\beta \quad & (\lambda x.x\ x)\ \underline{(\lambda x.x\ x)} \\ \rightarrow_\beta \quad & (\lambda x.x\ x)\ \underline{(\lambda x.x\ x)} \\ \rightarrow_\beta \quad & (\lambda x.x\ x)\ \underline{(\lambda x.x\ x)} \\ \ldots & \end{aligned}$$

In addition to showing the uniqueness of simplest (normal) forms, Church and Rosser showed that if any evaluation order will terminate, normal order will. This pair of results is known as the *Church-Rosser theorem*.

11.7.2 Control Flow

We noted at the beginning of the previous subsection that arithmetic can be modeled in the lambda calculus using a distinguished zero function (commonly the identity) and a successor function. What about control-flow constructs—selection and recursion in particular?

EXAMPLE 11.93
Booleans and conditionals

The select_first function, $\lambda x.\lambda y.x$, is commonly used to represent the Boolean value true. The select_second function, $\lambda x.\lambda y.y$, is commonly used to represent the Boolean value false. Let us denote these by T and F. The nice thing about these definitions is that they allow us to define an if function very easily:

$$\mathsf{if} \equiv \lambda c.\lambda t.\lambda e.c\ t\ e$$

Consider:

$$\begin{aligned}
\mathsf{if}\ T\ 3\ 4 &\equiv (\lambda c.\lambda t.\lambda e.c\ t\ e)\ (\lambda x.\lambda y.x)\ 3\ 4\\
&\to_\beta^* (\lambda x.\lambda y.x)\ 3\ 4\\
&\to_\beta^* 3
\end{aligned}$$

$$\begin{aligned}
\mathsf{if}\ F\ 3\ 4 &\equiv (\lambda c.\lambda t.\lambda e.c\ t\ e)\ (\lambda x.\lambda y.y)\ 3\ 4\\
&\to_\beta^* (\lambda x.\lambda y.y)\ 3\ 4\\
&\to_\beta^* 4
\end{aligned}$$

Functions like equal and greater_than can be defined to take numeric values as arguments, returning T or F.

EXAMPLE 11.94

Beta abstraction for recursion

Recursion is a little tricky. An equation like

$$\begin{aligned}
\mathsf{gcd} \equiv\ & \lambda a.\lambda b.(\mathsf{if}\ (\mathsf{equal}\ a\ b)\ a\\
& (\mathsf{if}\ (\mathsf{greater_than}\ a\ b)\ (\mathsf{gcd}\ (\mathsf{minus}\ a\ b)\ b)\ (\mathsf{gcd}\ (\mathsf{minus}\ b\ a)\ a)))
\end{aligned}$$

is not really a definition at all, because gcd appears on both sides. Our previous definitions (T, F, if) were simply shorthand: we could substitute them out to obtain a pure lambda expression. If we try that with gcd, the "definition" just gets bigger, with new occurrences of the gcd name. To obtain a real definition, we first rewrite our equation using *beta abstraction* (the opposite of beta reduction):

$$\begin{aligned}
\mathsf{gcd} \equiv\ & (\lambda g.\lambda a.\lambda b.(\mathsf{if}\ (\mathsf{equal}\ a\ b)\ a\\
& (\mathsf{if}\ (\mathsf{greater_than}\ a\ b)\ (g(\mathsf{minus}\ a\ b)\ b)\ (g(\mathsf{minus}\ b\ a)\ a))))\ \mathsf{gcd}
\end{aligned}$$

Now our equation has the form

$$\mathsf{gcd} \equiv f\ \mathsf{gcd}$$

where f is the perfectly well-defined (nonrecursive) lambda expression

$$\begin{aligned}
& \lambda g.\lambda a.\lambda b.(\mathsf{if}\ (\mathsf{equal}\ a\ b)\ a\\
& \quad (\mathsf{if}\ (\mathsf{greater_than}\ a\ b)\ (g\ (\mathsf{minus}\ a\ b)\ b)\ (g\ (\mathsf{minus}\ b\ a)\ a)))
\end{aligned}$$

Clearly gcd is a fixed point of f.

EXAMPLE 11.95

The fixed-point combinator **Y**

As it turns out, for any function f given by a lambda expression, we can find the least (simplest) fixed point of f, if there is a fixed point, by applying the *fixed-point combinator*

$$\lambda h.(\lambda x.h(xx))\ (\lambda x.h(xx))$$

commonly denoted **Y**. **Y** has the property that for any lambda expression f, if the normal-order evaluation of $\mathbf{Y}f$ terminates, then $f(\mathbf{Y}f)$ and $\mathbf{Y}f$ will reduce to the same simplest form (see Exercise C-11.21). In the case of our gcd function, we have

$$\begin{aligned}\text{gcd} \equiv\ & (\lambda h.(\lambda x.h(x\,x))\,(\lambda x.h(x\,x))) \\ & (\lambda g.\lambda a.\lambda b.(\text{if } (\text{equal } a\ b)\ a \\ & \quad (\text{if } (\text{greater\_than } a\ b)\ (g(\text{minus } a\ b)\ b)\ (g(\text{minus } b\ a)\ a))))\end{aligned}$$

Figure C-11.6 traces the evaluation of gcd 4 2. Given the existence of the **Y** combinator, most authors permit recursive "definitions" of functions, for convenience.

11.7.3 Structures

EXAMPLE 11.96
Lambda calculus list operators

Just as we can use functions to build numbers and truth values, we can also use them to encapsulate values in structures. Using Scheme terminology for the sake of clarity, we can define simple list-processing functions as follows:

$$\begin{aligned}\text{cons} &\equiv \lambda a.\lambda d.\lambda x.x\ a\ d \\ \text{car} &\equiv \lambda l.l\ \text{select\_first} \\ \text{cdr} &\equiv \lambda l.l\ \text{select\_second} \\ \text{nil} &\equiv \lambda x.T \\ \text{null?} &\equiv \lambda l.l(\lambda x.\lambda y.F)\end{aligned}$$

where select_first and select_second are the functions $\lambda x.\lambda y.x$ and $\lambda x.\lambda y.y$, respectively—functions we also use to represent true and false.

EXAMPLE 11.97
List operator identities

Using these definitions we can see that

$$\begin{aligned}\text{car}(\text{cons } A\ B) &\equiv (\lambda l.l\ \text{select\_first})\ (\text{cons } A\ B) \\ &\to_\beta (\text{cons } A\ B)\ \text{select\_first} \\ &\equiv ((\lambda a.\lambda d.\lambda x.x\ a\ d)\ A\ B)\ \text{select\_first} \\ &\to_\beta^* (\lambda x.x\ A\ B)\ \text{select\_first} \\ &\to_\beta \text{select\_first}\ A\ B \\ &\equiv (\lambda x.\lambda y.x)\ A\ B \\ &\to_\beta^* A\end{aligned}$$

$$\begin{aligned}\text{cdr}(\text{cons } A\ B) &\equiv (\lambda l.l\ \text{select\_second})\ (\text{cons } A\ B) \\ &\to_\beta (\text{cons } A\ B)\ \text{select\_second} \\ &\equiv ((\lambda a.\lambda d.\lambda x.x\ a\ d)\ A\ B)\ \text{select\_second} \\ &\to_\beta^* (\lambda x.x\ A\ B)\ \text{select\_second} \\ &\to_\beta \text{select\_second}\ A\ B \\ &\equiv (\lambda x.\lambda y.y)\ A\ B \\ &\to_\beta^* B\end{aligned}$$

$$
\begin{aligned}
\text{gcd } 2\,4 &\equiv \mathbf{Y} f\,2\,4 \\
&\equiv ((\lambda h.(\lambda x.h(x\,x))\,(\lambda x.h(x\,x)))f)\,2\,4 \\
&\to_\beta ((\lambda x.f(x\,x))\,(\lambda x.f(x\,x)))\,2\,4 \\
&\equiv (k\,k)\,2\,4, \quad \text{where } k \equiv \lambda x.f(x\,x) \\
&\to_\beta (f(k\,k))\,2\,4 \\
&\equiv ((\lambda g.\lambda a.\lambda b.(\text{if } (=\,a\,b)\,a\,(\text{if } (>\,a\,b)\,(g(-\,a\,b)\,b)\,(g(-\,b\,a)\,a))))\,(k\,k))\,2\,4 \\
&\to_\beta (\lambda a.\lambda b.(\text{if } (=\,a\,b)\,a\,(\text{if } (>\,a\,b)\,((k\,k)(-\,a\,b)\,b)\,((k\,k)(-\,b\,a)\,a))))\,2\,4 \\
&\to_\beta^* \text{if } (=\,2\,4)\,2\,(\text{if } (>\,2\,4)\,((k\,k)\,(-\,2\,4)\,4)\,((k\,k)\,(-\,4\,2)\,2)) \\
&\equiv (\lambda c.\lambda t.\lambda e.c\;t\;e)\,(=\,2\,4)\,2\,(\text{if } (>\,2\,4)\,((k\,k)\,(-\,2\,4)\,4)\,((k\,k)\,(-\,4\,2)\,2)) \\
&\to_\beta^* (=\,2\,4)\,2\,(\text{if } (>\,2\,4)\,((k\,k)\,(-\,2\,4)\,4)\,((k\,k)\,(-\,4\,2)\,2)) \\
&\to_\delta F\,2\,(\text{if } (>\,2\,4)\,((k\,k)\,(-\,2\,4)\,4)\,((k\,k)\,(-\,4\,2)\,2)) \\
&\equiv (\lambda x.\lambda y.y)\,2\,(\text{if } (>\,2\,4)\,((k\,k)\,(-\,2\,4)\,4)\,((k\,k)\,(-\,4\,2)\,2)) \\
&\to_\beta^* \text{if } (>\,2\,4)\,((k\,k)\,(-\,2\,4)\,4)\,((k\,k)\,(-\,4\,2)\,2) \\
&\to \ldots \\
&\to (k\,k)\,(-\,4\,2)\,2 \\
&\equiv ((\lambda x.f(x\,x))k)\,(-\,4\,2)\,2 \\
&\to_\beta (f(k\,k))\,(-\,4\,2)\,2 \\
&\equiv ((\lambda g.\lambda a.\lambda b.(\text{if } (=\,a\,b)\,a\,(\text{if } (>\,a\,b)\,(g(-\,a\,b)\,b)\,(g(-\,b\,a)\,a))))\,(k\,k))\,(-\,4\,2)\,2 \\
&\to_\beta (\lambda a.\lambda b.(\text{if } (=\,a\,b)\,a\,(\text{if } (>\,a\,b)\,((k\,k)(-\,a\,b)\,b)\,((k\,k)(-\,b\,a)\,a))))\,(-\,4\,2)\,2 \\
&\to_\beta^* \text{if } (=\,(-\,4\,2)\,2)\,(-\,4\,2)\,(\text{if } (>\,(-\,4\,2)\,2)\,((k\,k)\,(-\,(-\,4\,2)\,2)\,2)\,((k\,k)\,(-\,2\,(-\,4\,2))\,(-\,4\,2))) \\
&\equiv (\lambda c.\lambda t.\lambda e.c\;t\;e) \\
&\quad\; (=\,(-\,4\,2)\,2)\,(-\,4\,2)\,(\text{if } (>\,(-\,4\,2)\,2)\,((k\,k)\,(-\,(-\,4\,2)\,2)\,2)\,((k\,k)\,(-\,2\,(-\,4\,2))\,(-\,4\,2))) \\
&\to_\beta^* (=\,(-\,4\,2)\,2)\,(-\,4\,2)\,(\text{if } (>\,(-\,4\,2)\,2)\,((k\,k)\,(-\,(-\,4\,2)\,2)\,2)\,((k\,k)\,(-\,2\,(-\,4\,2))\,(-\,4\,2))) \\
&\to_\delta (=\,2\,2)\,(-\,4\,2)\,(\text{if } (>\,(-\,4\,2)\,2)\,((k\,k)\,(-\,(-\,4\,2)\,2)\,2)\,((k\,k)\,(-\,2\,(-\,4\,2))\,(-\,4\,2))) \\
&\to_\delta T\,(-\,4\,2)\,(\text{if } (>\,(-\,4\,2)\,2)\,((k\,k)\,(-\,(-\,4\,2)\,2)\,2)\,((k\,k)\,(-\,2\,(-\,4\,2))\,(-\,4\,2))) \\
&\equiv (\lambda x.\lambda y.x)\,(-\,4\,2)\,(\text{if } (>\,(-\,4\,2)\,2)\,((k\,k)\,(-\,(-\,4\,2)\,2)\,2)\,((k\,k)\,(-\,2\,(-\,4\,2))\,(-\,4\,2))) \\
&\to_\beta^* (-\,4\,2) \\
&\to_\delta 2
\end{aligned}
$$

Figure 11.6 **Evaluation of a recursive lambda expression.** As explained in the body of the text, gcd is defined to be the fixed-point combinator $\mathbf{Y}$ applied to a beta abstraction f of the standard recursive definition for greatest common divisor. Specifically, $\mathbf{Y}$ is $\lambda h.(\lambda x.h(x\,x))\,(\lambda x.h(x\,x))$ and f is $\lambda g.\lambda a.\lambda b.(\text{if } (=\,a\,b)\,a\,(\text{if } (>\,a\,b)\,(g(-\,a\,b)\,b)\,(g(-\,b\,a)\,a)))$. For brevity we have used =, >, and − in place of equal, greater_than, and minus. We have performed the evaluation in normal order.

$$\begin{aligned}
\text{null? nil} &\equiv (\lambda l.l\ (\lambda x.\lambda y.\text{select_second}))\ \text{nil} \\
&\to_\beta \text{nil}\ (\lambda x.\lambda y.\text{select_second}) \\
&\equiv (\lambda x.\text{select_first})\ (\lambda x.\lambda y.\text{select_second}) \\
&\to_\beta \text{select_first} \\
&\equiv T
\end{aligned}$$

$$\begin{aligned}
\text{null? (cons } A\ B) &\equiv (\lambda l.l\ (\lambda x.\lambda y.\text{select_second}))\ (\text{cons } A\ B) \\
&\to_\beta (\text{cons } A\ B)\ (\lambda x.\lambda y.\text{select_second}) \\
&\equiv ((\lambda a.\lambda d.\lambda x.x\ a\ d)\ A\ B)\ (\lambda x.\lambda y.\text{select_second}) \\
&\to_\beta^* (\lambda x.x\ A\ B)\ (\lambda x.\lambda y.\text{select_second}) \\
&\to_\beta (\lambda x.\lambda y.\text{select_second})\ A\ B \\
&\to_\beta^* \text{select_second} \\
&\equiv F
\end{aligned}$$

EXAMPLE 11.98
Nesting of lambda expressions

Because every lambda abstraction has a single argument, lambda expressions are naturally curried. We generally obtain the effect of a multiargument function by nesting lambda abstractions:

$$\text{compose} \equiv \lambda f.\lambda g.\lambda x.f\ (g\ x)$$

which groups as

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

We commonly think of compose as a function that takes two functions as arguments and returns a third function as its result. We could just as easily, however, think of compose as a function of three arguments: the f, g, and x above. The official story, or course, is that compose is a function of one argument that evaluates to a function of one argument that in turn evaluates to a function of one argument.

EXAMPLE 11.99
Paired arguments and currying

If desired, we can use our structure-building functions to define a noncurried version of compose whose (single) argument is a pair:

$$\text{paired\_compose} \equiv \lambda p.\lambda x.(\text{car } p)\ ((\text{cdr } p)\ x)$$

If we consider the pairing of arguments as a general technique, we can write a curry function that reproduces the single-argument version, just as we did in Scheme in Section 11.6:

$$\text{curry} \equiv \lambda f.\lambda a.\lambda b.f(\text{cons } a\ b)$$

CHECK YOUR UNDERSTANDING

29. What is the difference between *partial* and *total* functions? Why is the difference important?
30. What is meant by the *function space* $A \to B$?
31. Define *beta reduction*, *alpha conversion*, *eta reduction*, and *delta reduction*.
32. How does beta reduction in lambda calculus differ from lazy evaluation of arguments in a nonstrict programming language like Haskell?
33. Explain how lambda expressions can be used to represent Boolean values and control flow.
34. What is *beta abstraction*?
35. What is the **Y** combinator? What useful property does it possess?
36. Explain how lambda expressions can be used to represent structured values such as lists.
37. State the *Church-Rosser theorem*.

Functional Languages

11.10 Exercises

11.20 In Figure C-11.6 we evaluated our expression in normal order. Did we really have any choice? What would happen if we tried to use applicative order?

11.21 Prove that for any lambda expression f, if the normal-order evaluation of $\mathbf{Y}f$ terminates, where $\mathbf{Y}$ is the fixed-point combinator $\lambda h.(\lambda x.h(x\,x))\,(\lambda x.h(x\,x))$, then $f(\mathbf{Y}f)$ and $\mathbf{Y}f$ will reduce to the same simplest form.

11.22 Given the definition of structures (lists) in Section C-11.7.3, what happens if we apply car or cdr to nil? How might you introduce the notion of "type error" into lambda calculus?

11.23 Let

$$\text{zero} \equiv \lambda x.x$$
$$\text{succ} \equiv \lambda n.(\lambda s.(s\ \text{select\_second})\ n)$$

where $\text{select\_second} \equiv \lambda x.\lambda y.y$. Now let

$$\text{one} \equiv \text{succ zero}$$
$$\text{two} \equiv \text{succ one}$$

Show that

$$\text{one select\_second} = \text{zero}$$
$$\text{two select\_second select\_second} = \text{zero}$$

In general, show that

$$\text{succ}^n\ \text{zero select\_second}^n = \text{zero}$$

Use this result to define a predecessor function pred. You may ignore the issue of the predecessor of zero.

Note that our definitions of T and F allow us to check whether a number is equal to zero:

$$\text{iszero} \equiv \lambda n.(n\ \text{select\_first})$$

Using succ, pred, iszero, and if, show how to define plus and times recursively. These definitions could of course be made nonrecursive by means of beta abstraction and **Y**.

Functional Languages

11.11 Explorations

11.30 Learn about the *typed lambda calculus*. What properties does it have that standard lambda calculus does not? What restrictions does it place on permissible expressions? Possible places to start include Cardelli and Wegner's classic survey [CW85] or the newer text by Pierce [Pie02].

11.31 Learn more about *fixed points*. We mentioned these when presenting the **Y** combinator in Section C-11.7.2. They also arise in the denotational definition of loop constructs, in metacircular interpreters [AS96, Sec. 4.1]), and in the *data flow analysis* used by optimizing compilers (Section C-17.4.2). What do these subjects have in common? Are there important differences as well?

11.32 Explore the connection between lexical scoping in Scheme or OCaml and the notion of free and bound variables in lambda calculus. How closely are these related? Why does lambda calculus require alpha conversion but Scheme and OCaml do not? Is there any analogy in lambda calculus to the dynamic scoping of early dialects of Lisp?