The lambda-calculus can be enriched in a variety of ways. First, it is often convenient to add special concrete syntax for features like numbers, tuples, records, etc., whose behavior can already be simulated in the core language. More interestingly, we can add more complex features such as mutable reference cells or nonlocal exception handling, which can be modeled in the core language only by using rather heavy translations. Such extensions lead eventually to languages such as ML (Gordon, Milner, and Wadsworth, 1979; Milner, Tofte, and Harper, 1990; Weis, Aponte, Laville, Mauny, and Suárez, 1989; Milner, Tofte, Harper, and MacQueen, 1997), Haskell (Hudak et al., 1992), or Scheme (Sussman and Steele, 1975; Kelsey, Clinger, and Rees, 1998). As we shall see in later chapters, extensions to the core language often involve extensions to the type system as well.

5.1 Basics

Procedural (or functional) abstraction is a key feature of essentially all programming languages. Instead of writing the same calculation over and over, we write a procedure or function that performs the calculation generically, in terms of one or more named parameters, and then instantiate this function as needed, providing values for the parameters in each case. For example, it is second nature for a programmer to take a long and repetitive expression like

```
(5*4*3*2*1) + (7*6*5*4*3*2*1) - (3*2*1)
```

and rewrite it as `factorial(5) + factorial(7) - factorial(3)`, where:

```
factorial(n)  =  if n=0 then 1 else n * factorial(n-1).
```

For each nonnegative number `n`, instantiating the function `factorial` with the argument `n` yields the factorial of `n` as result. If we write "`λn. ...`" as a shorthand for "the function that, for each `n`, yields...," we can restate the definition of `factorial` as:

```
factorial  =  λn. if n=0 then 1 else n * factorial(n-1)
```

Then `factorial(0)` means "the function `(λn. if n=0 then 1 else ...)` applied to the argument `0`," that is, "the value that results when the argument variable `n` in the function body `(λn. if n=0 then 1 else ...)` is replaced by `0`," that is, "`if 0=0 then 1 else ...`," that is, `1`.

The *lambda-calculus* (or λ-calculus) embodies this kind of function definition and application in the purest possible form. In the lambda-calculus *everything* is a function: the arguments accepted by functions are themselves functions and the result returned by a function is another function.

The syntax of the lambda-calculus comprises just three sorts of terms.[1] A variable `x` by itself is a term; the abstraction of a variable `x` from a term t_1, written $\lambda x.t_1$, is a term; and the application of a term t_1 to another term t_2, written $t_1\ t_2$, is a term. These ways of forming terms are summarized in the following grammar.

t ::=		*terms:*
	x	*variable*
	λx.t	*abstraction*
	t t	*application*

The subsections that follow explore some fine points of this definition.

Abstract and Concrete Syntax

When discussing the syntax of programming languages, it is useful to distinguish two levels[2] of structure. The *concrete syntax* (or *surface syntax*) of the language refers to the strings of characters that programmers directly read and write. *Abstract syntax* is a much simpler internal representation of programs as labeled trees (called *abstract syntax trees* or *ASTs*). The tree representation renders the structure of terms immediately obvious, making it a natural fit for the complex manipulations involved in both rigorous language definitions (and proofs about them) and the internals of compilers and interpreters.

The transformation from concrete to abstract syntax takes place in two stages. First, a *lexical analyzer* (or *lexer*) converts the string of characters written by the programmer into a sequence of *tokens*—identifiers, keywords, constants, punctuation, etc. The lexer removes comments and deals with issues such as whitespace and capitalization conventions, and formats for numeric and string constants. Next, a *parser* transforms this sequence of tokens into an abstract syntax tree. During parsing, various conventions such as operator *precedence* and *associativity* reduce the need to clutter surface programs with parentheses to explicitly indicate the structure of compound expressions. For example, * binds more tightly than +, so the parser interprets the unparen-

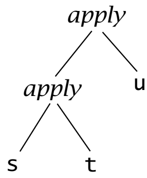
1. The phrase *lambda-term* is used to refer to arbitrary terms in the lambda-calculus. Lambda-terms beginning with a λ are often called *lambda-abstractions*.

2. Definitions of full-blown languages sometimes use even more levels. For example, following Landin, it is often useful to define the behaviors of some languages constructs as derived forms, by translating them into combinations of other, more basic, features. The restricted sublanguage containing just these core features is then called the *internal language* (or *IL*), while the full language including all derived forms is called the *external language* (*EL*). The transformation from EL to IL is (at least conceptually) performed in a separate pass, following parsing. Derived forms are discussed in Section 11.3.

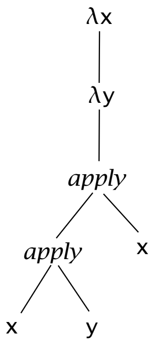
thesized expression `1+2*3` as the abstract syntax tree to the left below rather than the one to the right:

The focus of attention in this book is on abstract, not concrete, syntax. Grammars like the one for lambda-terms above should be understood as describing legal tree structures, not strings of tokens or characters. Of course, when we write terms in examples, definitions, theorems, and proofs, we will need to express them in a concrete, linear notation, but we always have their underlying abstract syntax trees in mind.

To save writing too many parentheses, we adopt two conventions when writing lambda-terms in linear form. First, application associates to the left—that is, `s t u` stands for the same tree as `(s t) u`:



Second, the bodies of abstractions are taken to extend as far to the right as possible, so that, for example, `λx. λy. x y x` stands for the same tree as `λx. (λy. ((x y) x))`:



Variables and Metavariables

Another subtlety in the syntax definition above concerns the use of metavariables. We will continue to use the metavariable `t` (as well as `s`, and `u`, with or without subscripts) to stand for an arbitrary term.[3] Similarly, x (as well as y and z) stands for an arbitrary variable. Note, here, that x is a metavariable ranging over variables! To make matters worse, the set of short names is limited, and we will also want to use x, y, etc. as object-language variables. In such cases, however, the context will always make it clear which is which. For example, in a sentence like "The term λx. λy. x y has the form λz.s, where z = x and s = λy. x y," the names z and s are metavariables, whereas x and y are object-language variables.

Scope

A final point we must address about the syntax of the lambda-calculus is the *scopes* of variables.

An occurrence of the variable x is said to be *bound* when it occurs in the body t of an abstraction λx.t. (More precisely, it is bound by *this* abstraction. Equivalently, we can say that λx is a *binder* whose scope is t.) An occurrence of x is *free* if it appears in a position where it is not bound by an enclosing abstraction on x. For example, the occurrences of x in x y and λy. x y are free, while the ones in λx.x and λz. λx. λy. x (y z) are bound. In (λx.x) x, the first occurrence of x is bound and the second is free.

A term with no free variables is said to be *closed;* closed terms are also called *combinators*. The simplest combinator, called the *identity function,*

```
id = λx.x;
```

does nothing but return its argument.

Operational Semantics

In its pure form, the lambda-calculus has no built-in constants or primitive operators—no numbers, arithmetic operations, conditionals, records, loops, sequencing, I/O, etc. The sole means by which terms "compute" is the application of functions to arguments (which themselves are functions). Each step in the computation consists of rewriting an application whose left-hand component is an abstraction, by substituting the right-hand component for the bound variable in the abstraction's body. Graphically, we write

$$(\lambda x.\ t_{12})\ t_2 \longrightarrow [x \mapsto t_2]t_{12},$$

where $[x \mapsto t_2]t_{12}$ means "the term obtained by replacing all free occurrences of x in t_{12} by t_2." For example, the term (λx.x) y evaluates to y and

3. Naturally, in this chapter, t ranges over lambda-terms, not arithmetic expressions. Throughout the book, t will always range over the terms of calculus under discussion at the moment. A footnote on the first page of each chapter specifies which system this is.

the term `(λx. x (λx.x)) (u r)` evaluates to `u r (λx.x)`. Following Church, a term of the form $(\lambda\mathtt{x}.\ \mathtt{t}_{12})\ \mathtt{t}_2$ is called a *redex* ("reducible expression"), and the operation of rewriting a redex according to the above rule is called *beta-reduction.*

Several different evaluation strategies for the lambda-calculus have been studied over the years by programming language designers and theorists. Each strategy defines which redex or redexes in a term can fire on the next step of evaluation.[4]

- Under *full beta-reduction,* any redex may be reduced at any time. At each step we pick some redex, anywhere inside the term we are evaluating, and reduce it. For example, consider the term

  `(λx.x) ((λx.x) (λz. (λx.x) z)),`

  which we can write more readably as `id (id (λz. id z))`. This term contains three redexes:

  $$\begin{array}{l} \underline{\mathtt{id\ (id\ (}\lambda\mathtt{z.\ id\ z))}} \\ \mathtt{id\ (}\underline{\mathtt{(id\ (}\lambda\mathtt{z.\ id\ z))}}\mathtt{)} \\ \mathtt{id\ (id\ (}\lambda\mathtt{z.\ }\underline{\mathtt{id\ z}}\mathtt{))} \end{array}$$

  Under full beta-reduction, we might choose, for example, to begin with the innermost redex, then do the one in the middle, then the outermost:

  $$\begin{array}{ll} & \mathtt{id\ (id\ (}\lambda\mathtt{z.\ }\underline{\mathtt{id\ z}}\mathtt{))} \\ \longrightarrow & \mathtt{id\ }\underline{\mathtt{(id\ (}\lambda\mathtt{z.z))}} \\ \longrightarrow & \underline{\mathtt{id\ (}\lambda\mathtt{z.z)}} \\ \longrightarrow & \lambda\mathtt{z.z} \\ \not\longrightarrow & \end{array}$$

- Under the *normal order* strategy, the leftmost, outermost redex is always reduced first. Under this strategy, the term above would be reduced as follows:

  $$\begin{array}{ll} & \underline{\mathtt{id\ (id\ (}\lambda\mathtt{z.\ id\ z))}} \\ \longrightarrow & \underline{\mathtt{id\ (}\lambda\mathtt{z.\ id\ z)}} \\ \longrightarrow & \lambda\mathtt{z.\ }\underline{\mathtt{id\ z}} \\ \longrightarrow & \lambda\mathtt{z.z} \\ \not\longrightarrow & \end{array}$$

4. Some people use the terms "reduction" and "evaluation" synonymously. Others use "evaluation" only for strategies that involve some notion of "value" and "reduction" otherwise.

Under this strategy (and the ones below), the evaluation relation is actually a partial function: each term `t` evaluates in one step to at most one term `t′`.

- The *call by name* strategy is yet more restrictive, allowing no reductions inside abstractions. Starting from the same term, we would perform the first two reductions as under normal-order, but then stop before the last and regard $\lambda\texttt{z. id z}$ as a normal form:

$$
\begin{array}{rl}
 & \underline{\texttt{id (id (}\lambda\texttt{z. id z))}} \\
\longrightarrow & \underline{\texttt{id (}\lambda\texttt{z. id z)}} \\
\longrightarrow & \lambda\texttt{z. id z} \\
\not\longrightarrow &
\end{array}
$$

  Variants of call by name have been used in some well-known programming languages, notably Algol-60 (Naur et al., 1963) and Haskell (Hudak et al., 1992). Haskell actually uses an optimized version known as *call by need* (Wadsworth, 1971; Ariola et al., 1995) that, instead of re-evaluating an argument each time it is used, overwrites all occurrences of the argument with its value the first time it is evaluated, avoiding the need for subsequent re-evaluation. This strategy demands that we maintain some sharing in the run-time representation of terms—in effect, it is a reduction relation on abstract syntax *graphs,* rather than syntax trees.

- Most languages use a *call by value* strategy, in which only outermost redexes are reduced *and* where a redex is reduced only when its right-hand side has already been reduced to a *value*—a term that is finished computing and cannot be reduced any further.[5] Under this strategy, our example term reduces as follows:

$$
\begin{array}{rl}
 & \texttt{id }\underline{\texttt{(id (}\lambda\texttt{z. id z))}} \\
\longrightarrow & \underline{\texttt{id (}\lambda\texttt{z. id z)}} \\
\longrightarrow & \lambda\texttt{z. id z} \\
\not\longrightarrow &
\end{array}
$$

  The call-by-value strategy is *strict*, in the sense that the arguments to functions are always evaluated, whether or not they are used by the body of the function. In contrast, *non-strict* (or *lazy*) strategies such as call-by-name and call-by-need evaluate only the arguments that are actually used.

5. In the present bare-bones calculus, the only values are lambda-abstractions. Richer calculi will include other values: numeric and boolean constants, strings, tuples of values, records of values, lists of values, etc.

The choice of evaluation strategy actually makes little difference when discussing type systems. The issues that motivate various typing features, and the techniques used to address them, are much the same for all the strategies. In this book, we use call by value, both because it is found in most well-known languages and because it is the easiest to enrich with features such as exceptions (Chapter 14) and references (Chapter 13).

5.2 Programming in the Lambda-Calculus

The lambda-calculus is much more powerful than its tiny definition might suggest. In this section, we develop a number of standard examples of programming in the lambda-calculus. These examples are not intended to suggest that the lambda-calculus should be taken as a full-blown programming language in its own right—all widely used high-level languages provide clearer and more efficient ways of accomplishing the same tasks—but rather are intended as warm-up exercises to get the feel of the system.

Multiple Arguments

To begin, observe that the lambda-calculus provides no built-in support for multi-argument functions. Of course, this would not be hard to add, but it is even easier to achieve the same effect using *higher-order functions* that yield functions as results. Suppose that `s` is a term involving two free variables `x` and `y` and that we want to write a function `f` that, for each pair `(v,w)` of arguments, yields the result of substituting `v` for `x` and `w` for `y` in `s`. Instead of writing `f = λ(x,y).s`, as we might in a richer programming language, we write `f = λx.λy.s`. That is, `f` is a function that, given a value `v` for `x`, yields a function that, given a value `w` for `y`, yields the desired result. We then apply `f` to its arguments one at a time, writing `f v w` (i.e., `(f v) w`), which reduces to `((λy.[x ↦ v]s) w)` and thence to `[y ↦ w][x ↦ v]s`. This transformation of multi-argument functions into higher-order functions is called *currying* in honor of Haskell Curry, a contemporary of Church.

Church Booleans

Another language feature that can easily be encoded in the lambda-calculus is boolean values and conditionals. Define the terms `tru` and `fls` as follows:

```
tru = λt. λf. t;
fls = λt. λf. f;
```

(The abbreviated spellings of these names are intended to help avoid confusion with the primitive boolean constants `true` and `false` from Chapter 3.)

The terms `tru` and `fls` can be viewed as *representing* the boolean values "true" and "false," in the sense that we can use these terms to perform the operation of testing the truth of a boolean value. In particular, we can use application to define a combinator `test` with the property that `test b v w` reduces to `v` when `b` is `tru` and reduces to `w` when `b` is `fls`.

```
test = λl. λm. λn. l m n;
```

The `test` combinator does not actually do much: `test b v w` just reduces to `b v w`. In effect, the boolean `b` itself is the conditional: it takes two arguments and chooses the first (if it is `tru`) or the second (if it is `fls`). For example, the term `test tru v w` reduces as follows:

$$
\begin{array}{lll}
 & \texttt{test tru v w} & \\
= & \underline{\texttt{(λl. λm. λn. l m n) tru}}\ \texttt{v w} & \text{by definition} \\
\longrightarrow & \underline{\texttt{(λm. λn. tru m n) v}}\ \texttt{w} & \text{reducing the underlined redex} \\
\longrightarrow & \underline{\texttt{(λn. tru v n) w}} & \text{reducing the underlined redex} \\
\longrightarrow & \texttt{tru v w} & \text{reducing the underlined redex} \\
= & \underline{\texttt{(λt.λf.t) v}}\ \texttt{w} & \text{by definition} \\
\longrightarrow & \underline{\texttt{(λf. v) w}} & \text{reducing the underlined redex} \\
\longrightarrow & \texttt{v} & \text{reducing the underlined redex}
\end{array}
$$

We can also define boolean operators like logical conjunction as functions:

```
and = λb. λc. b c fls;
```

That is, `and` is a function that, given two boolean values `b` and `c`, returns `c` if `b` is `tru` and `fls` if `b` is `fls`; thus `and b c` yields `tru` if both `b` and `c` are `tru` and `fls` if either `b` or `c` is `fls`.

```
and tru tru;
```

▸ `(λt. λf. t)`

```
and tru fls;
```

▸ `(λt. λf. f)`

5.2.1 EXERCISE [★]: Define logical `or` and `not` functions. □

Pairs

Using booleans, we can encode pairs of values as terms.

```
pair = λf.λs.λb. b f s;
fst = λp. p tru;
snd = λp. p fls;
```

That is, `pair v w` is a function that, when applied to a boolean value `b`, applies `b` to `v` and `w`. By the definition of booleans, this application yields `v` if `b` is `tru` and `w` if `b` is `fls`, so the first and second projection functions `fst` and `snd` can be implemented simply by supplying the appropriate boolean. To check that `fst (pair v w)` $\longrightarrow^*$ `v`, calculate as follows:

	$\mathtt{fst\ (pair\ v\ w)}$	
$=$	$\mathtt{fst}\ (\underline{(\lambda f.\ \lambda s.\ \lambda b.\ b\ f\ s)\ v}\ w)$	by definition
$\longrightarrow$	$\mathtt{fst}\ (\underline{(\lambda s.\ \lambda b.\ b\ v\ s)\ w})$	reducing the underlined redex
$\longrightarrow$	$\mathtt{fst}\ (\lambda b.\ b\ v\ w)$	reducing the underlined redex
$=$	$\underline{(\lambda p.\ p\ \mathtt{tru})\ (\lambda b.\ b\ v\ w)}$	by definition
$\longrightarrow$	$\underline{(\lambda b.\ b\ v\ w)\ \mathtt{tru}}$	reducing the underlined redex
$\longrightarrow$	$\mathtt{tru}\ v\ w$	reducing the underlined redex
$\longrightarrow^*$	v	as before.

Church Numerals

Representing numbers by lambda-terms is only slightly more intricate than what we have just seen. Define the *Church numerals* c_0, c_1, c_2, etc., as follows:

```
c0 = λs. λz. z;
c1 = λs. λz. s z;
c2 = λs. λz. s (s z);
c3 = λs. λz. s (s (s z));
etc.
```

That is, each number *n* is represented by a combinator c_n that takes two arguments, `s` and `z` (for "successor" and "zero"), and applies `s`, *n* times, to `z`. As with booleans and pairs, this encoding makes numbers into active entities: the number *n* is represented by a function that does something *n* times—a kind of active unary numeral.

(The reader may already have observed that c_0 and `fls` are actually the same term. Similar "puns" are common in assembly languages, where the same pattern of bits may represent many different values—an int, a float, an address, four characters, etc.—depending on how it is interpreted, and in low-level languages such as C, which also identifies `0` and `false`.)

We can define the successor function on Church numerals as follows:

```
scc = λn. λs. λz. s (n s z);
```

The term `scc` is a combinator that takes a Church numeral `n` and returns another Church numeral—that is, it yields a function that takes arguments `s` and `z` and applies `s` repeatedly to `z`. We get the right number of applications of `s` to `z` by first passing `s` and `z` as arguments to `n`, and then explicitly applying `s` one more time to the result.

5.2.2 EXERCISE [★★]: Find another way to define the successor function on Church numerals. □

Similarly, addition of Church numerals can be performed by a term `plus` that takes two Church numerals, `m` and `n`, as arguments, and yields another Church numeral—i.e., a function—that accepts arguments `s` and `z`, applies `s` iterated `n` times to `z` (by passing `s` and `z` as arguments to `n`), and then applies `s` iterated `m` more times to the result:

```
plus = λm. λn. λs. λz. m s (n s z);
```

The implementation of multiplication uses another trick: since `plus` takes its arguments one at a time, applying it to just one argument `n` yields the function that adds `n` to whatever argument it is given. Passing this function as the first argument to `m` and c_0 as the second argument means "apply the function that adds `n` to its argument, iterated `m` times, to zero," i.e., "add together `m` copies of `n`."

```
times = λm. λn. m (plus n) c₀;
```

5.2.3 EXERCISE [★★]: Is it possible to define multiplication on Church numerals without using `plus`? □

5.2.4 EXERCISE [RECOMMENDED, ★★]: Define a term for raising one number to the power of another. □

To test whether a Church numeral is zero, we must find some appropriate pair of arguments that will give us back this information—specifically, we must apply our numeral to a pair of terms `zz` and `ss` such that applying `ss` to `zz` one or more times yields `fls`, while not applying it at all yields `tru`. Clearly, we should take `zz` to be just `tru`. For `ss`, we use a function that throws away its argument and always returns `fls`:

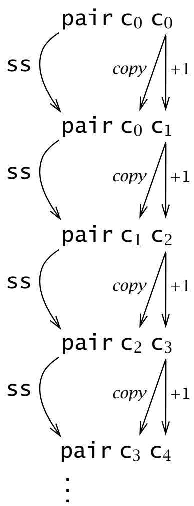


Figure 5-1: The predecessor function's "inner loop"

```
  iszro = λm. m (λx. fls) tru;

  iszro c1;
▸ (λt. λf. f)

  iszro (times c0 c2);
▸ (λt. λf. t)
```

Surprisingly, subtraction using Church numerals is quite a bit more difficult than addition. It can be done using the following rather tricky "predecessor function," which, given c_0 as argument, returns c_0 and, given c_{i+1}, returns c_i:

```
zz = pair c0 c0;
ss = λp. pair (snd p) (plus c1 (snd p));
prd = λm. fst (m ss zz);
```

This definition works by using `m` as a function to apply `m` copies of the function `ss` to the starting value `zz`. Each copy of `ss` takes a pair of numerals `pair` c_i c_j as its argument and yields `pair` c_j c_{j+1} as its result (see Figure 5-1). So applying `ss`, `m` times, to `pair` c_0 c_0 yields `pair` c_0 c_0 when $m = 0$ and `pair` c_{m-1} c_m when `m` is positive. In both cases, the predecessor of `m` is found in the first component.

5.2.5 EXERCISE [★★]: Use `prd` to define a subtraction function. □

5.2.6 EXERCISE [★★]: Approximately how many steps of evaluation (as a function of n) are required to calculate `prd` c_n? □

5.2.7 EXERCISE [★★]: Write a function `equal` that tests two numbers for equality and returns a Church boolean. For example,

```
  equal c₃ c₃;
▸ (λt. λf. t)

  equal c₃ c₂;
▸ (λt. λf. f)
```
□

Other common datatypes like lists, trees, arrays, and variant records can be encoded using similar techniques.

5.2.8 EXERCISE [RECOMMENDED, ★★★]: A list can be represented in the lambda-calculus by its `fold` function. (OCaml's name for this function is `fold_left`; it is also sometimes called `reduce`.) For example, the list `[x,y,z]` becomes a function that takes two arguments `c` and `n` and returns `c x (c y (c z n)))`. What would the representation of `nil` be? Write a function `cons` that takes an element `h` and a list (that is, a fold function) `t` and returns a similar representation of the list formed by prepending `h` to `t`. Write `isnil` and `head` functions, each taking a list parameter. Finally, write a `tail` function for this representation of lists (this is quite a bit harder and requires a trick analogous to the one used to define `prd` for numbers). □

Enriching the Calculus

We have seen that booleans, numbers, and the operations on them can be encoded in the pure lambda-calculus. Indeed, strictly speaking, we can do all the programming we ever need to without going outside of the pure system. However, when working with examples it is often convenient to include the primitive booleans and numbers (and possibly other data types) as well. When we need to be clear about precisely which system we are working in, we will use the symbol λ for the pure lambda-calculus as defined in Figure 5-3 and **λNB** for the enriched system with booleans and arithmetic expressions from Figures 3-1 and 3-2.

In **λNB**, we actually have two different implementations of booleans and two of numbers to choose from when writing programs: the real ones and the encodings we've developed in this chapter. Of course, it is easy to convert back and forth between the two. To turn a Church boolean into a primitive boolean, we apply it to `true` and `false`:

```
realbool = λb. b true false;
```

To go the other direction, we use an `if` expression:

```
churchbool = λb. if b then tru else fls;
```

We can build these conversions into higher-level operations. Here is an equality function on Church numerals that returns a real boolean:

```
realeq = λm. λn. (equal m n) true false;
```

In the same way, we can convert a Church numeral into the corresponding primitive number by applying it to `succ` and `0`:

```
realnat = λm. m (λx. succ x) 0;
```

We cannot apply `m` to `succ` directly, because `succ` by itself does not make syntactic sense: the way we defined the syntax of arithmetic expressions, `succ` must always be applied to something. We work around this by packaging `succ` inside a little function that does nothing but return the `succ` of its argument.

The reasons that primitive booleans and numbers come in handy for examples have to do primarily with evaluation order. For instance, consider the term `scc` c_1. From the discussion above, we might expect that this term should evaluate to the Church numeral c_2. In fact, it does not:

```
scc c1;
▸ (λs. λz. s ((λs'. λz'. s' z') s z))
```

This term contains a redex that, if we were to reduce it, would bring us (in two steps) to c_2, but the rules of call-by-value evaluation do not allow us to reduce it yet, since it is under a lambda-abstraction.

There is no fundamental problem here: the term that results from evaluation of `scc` c_1 is obviously *behaviorally equivalent* to c_2, in the sense that applying it to any pair of arguments `v` and `w` will yield the same result as applying c_2 to `v` and `w`. Still, the leftover computation makes it a bit difficult to check that our `scc` function is behaving the way we expect it to. For more complicated arithmetic calculations, the difficulty is even worse. For example, `times` c_2 c_2 evaluates not to c_4 but to the following monstrosity:

```
times c2 c2;
▸ (λs.
    λz.
      (λs'. λz'. s' (s' z')) s
      ((λs'.
```

```
            λz'.
              (λs". λz". s" (s" z")) s'
              ((λs". λz".z") s' z'))
         s
         z))
```

One way to check that this term behaves like c_4 is to test them for equality:

```
equal c₄ (times c₂ c₂);
▸ (λt. λf. t)
```

But it is more direct to take `times` c_2 c_2 and convert it to a primitive number:

```
realnat (times c₂ c₂);
▸ 4
```

The conversion has the effect of supplying the two extra arguments that `times` c_2 c_2 is waiting for, forcing all of the latent computation in its body.

Recursion

Recall that a term that cannot take a step under the evaluation relation is called a *normal form*. Interestingly, some terms cannot be evaluated to a normal form. For example, the *divergent* combinator

```
omega = (λx. x x) (λx. x x);
```

contains just one redex, and reducing this redex yields exactly `omega` again! Terms with no normal form are said to *diverge*.

The `omega` combinator has a useful generalization called the *fixed-point combinator*,[6] which can be used to help define recursive functions such as `factorial`.[7]

```
fix = λf. (λx. f (λy. x x y)) (λx. f (λy. x x y));
```

Like `omega`, the `fix` combinator has an intricate, repetitive structure; it is difficult to understand just by reading its definition. Probably the best way of getting some intuition about its behavior is to watch how it works on a specific example.[8] Suppose we want to write a recursive function definition

6. It is often called the *call-by-value Y-combinator*. Plotkin (1975) called it Z.

7. Note that the simpler call-by-name fixed point combinator

```
Y = λf. (λx. f (x x)) (λx. f (x x))
```

is useless in a call-by-value setting, since the expression `Y g` diverges, for any `g`.

8. It is also possible to derive the definition of `fix` from first principles (e.g., Friedman and Felleisen, 1996, Chapter 9), but such derivations are also fairly intricate.

of the form h = ⟨*body containing* h⟩—i.e., we want to write a definition where the term on the right-hand side of the = uses the very function that we are defining, as in the definition of factorial on page 52. The intention is that the recursive definition should be "unrolled" at the point where it occurs; for example, the definition of factorial would intuitively be

```
if n=0 then 1
else n * (if n-1=0 then 1
          else (n-1) * (if (n-2)=0 then 1
                        else (n-2) * ...))
```

or, in terms of Church numerals:

```
if realeq n c₀ then c₁
else times n (if realeq (prd n) c₀ then c₁
              else times (prd n)
                         (if realeq (prd (prd n)) c₀ then c₁
                          else times (prd (prd n)) ...))
```

This effect can be achieved using the fix combinator by first defining g = λf.⟨*body containing* f⟩ and then h = fix g. For example, we can define the factorial function by

```
g = λfct. λn. if realeq n c₀ then c₁ else (times n (fct (prd n)));
factorial = fix g;
```

Figure 5-2 shows what happens to the term factorial c_3 during evaluation. The key fact that makes this calculation work is that fct n $\longrightarrow^*$ g fct n. That is, fct is a kind of "self-replicator" that, when applied to an argument, supplies *itself* and n as arguments to g. Wherever the first argument to g appears in the body of g, we will get another copy of fct, which, when applied to an argument, will again pass itself and that argument to g, etc. Each time we make a recursive call using fct, we unroll one more copy of the body of g and equip it with new copies of fct that are ready to do the unrolling again.

5.2.9 Exercise [★]: Why did we use a primitive if in the definition of g, instead of the Church-boolean test function on Church booleans? Show how to define the factorial function in terms of test rather than if. □

5.2.10 Exercise [★★]: Define a function churchnat that converts a primitive natural number into the corresponding Church numeral. □

5.2.11 Exercise [Recommended, ★★]: Use fix and the encoding of lists from Exercise 5.2.8 to write a function that sums lists of Church numerals. □

$$
\begin{array}{ll}
 & \texttt{factorial } c_3 \\
= & \texttt{fix g } c_3 \\
\longrightarrow & \texttt{h h } c_3 \\
 & \text{where } \texttt{h} = \lambda\texttt{x. g } (\lambda\texttt{y. x x y}) \\
\longrightarrow & \texttt{g fct } c_3 \\
 & \text{where } \texttt{fct} = \lambda\texttt{y. h h y} \\
\longrightarrow & (\lambda\texttt{n. if realeq n } c_0 \\
 & \qquad\quad \texttt{then } c_1 \\
 & \qquad\quad \texttt{else times n (fct (prd n)))} \\
 & \qquad c_3 \\
\longrightarrow & \texttt{if realeq } c_3\ c_0 \\
 & \quad \texttt{then } c_1 \\
 & \quad \texttt{else times } c_3 \texttt{ (fct (prd } c_3\texttt{))} \\
\longrightarrow^* & \texttt{times } c_3 \texttt{ (fct (prd } c_3\texttt{))} \\
\longrightarrow^* & \texttt{times } c_3 \texttt{ (fct } c_2'\texttt{)} \\
 & \text{where } c_2' \text{ is behaviorally equivalent to } c_2 \\
\longrightarrow^* & \texttt{times } c_3 \texttt{ (g fct } c_2'\texttt{)} \\
\longrightarrow^* & \texttt{times } c_3 \texttt{ (times } c_2' \texttt{ (g fct } c_1'\texttt{))}. \\
 & \text{where } c_1' \text{ is behaviorally equivalent to } c_1 \\
 & \text{(by repeating the same calculation for } \texttt{g fct } c_2'\text{)} \\
\longrightarrow^* & \texttt{times } c_3 \texttt{ (times } c_2' \texttt{ (times } c_1' \texttt{ (g fct } c_0'\texttt{)))}. \\
 & \text{where } c_0' \text{ is behaviorally equivalent to } c_0 \\
 & \text{(similarly)} \\
\longrightarrow^* & \texttt{times } c_3 \texttt{ (times } c_2' \texttt{ (times } c_1' \texttt{ (if realeq } c_0'\ c_0 \texttt{ then } c_1 \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{else ...)))} \\
\longrightarrow^* & \texttt{times } c_3 \texttt{ (times } c_2' \texttt{ (times } c_1'\ c_1\texttt{))} \\
\longrightarrow^* & c_6' \\
 & \text{where } c_6' \text{ is behaviorally equivalent to } c_6.
\end{array}
$$

Figure 5-2: Evaluation of `factorial` c_3

Representation

Before leaving our examples behind and proceeding to the formal definition of the lambda-calculus, we should pause for one final question: What, exactly, does it mean to say that the Church numerals *represent* ordinary numbers?

To answer, we first need to remind ourselves of what the ordinary numbers are. There are many (equivalent) ways to define them; the one we have chosen here (in Figure 3-2) is to give:

- a constant 0,

- an operation `iszero` mapping numbers to booleans, and
- two operations, `succ` and `pred`, mapping numbers to numbers.

The behavior of the arithmetic operations is defined by the evaluation rules in Figure 3-2. These rules tell us, for example, that 3 is the successor of 2, and that `iszero 0` is true.

The Church encoding of numbers represents each of these elements as a lambda-term (i.e., a function):

- The term c_0 represents the number 0.

  As we saw on page 64, there are also "non-canonical representations" of numbers as terms. For example, λs. λz. (λx. x) z, which is behaviorally equivalent to c_0, also represents 0.

- The terms `scc` and `prd` represent the arithmetic operations `succ` and `pred`, in the sense that, if `t` is a representation of the number n, then `scc t` evaluates to a representation of $n + 1$ and `prd t` evaluates to a representation of $n - 1$ (or of 0, if n is 0).

- The term `iszro` represents the operation `iszero`, in the sense that, if `t` is a representation of 0, then `iszro t` evaluates to `true`,[9] and if `t` represents any number other than 0, then `iszro t` evaluates to `false`.

Putting all this together, suppose we have a whole program that does some complicated calculation with numbers to yield a boolean result. If we replace all the numbers and arithmetic operations with lambda-terms representing them and evaluate the program, we will get the same result. Thus, in terms of their effects on the overall results of programs, there is no observable difference between the real numbers and their Church-numeral representation.

5.3 Formalities

For the rest of the chapter, we consider the syntax and operational semantics of the lambda-calculus in more detail. Most of the structure we need is closely analogous to what we saw in Chapter 3 (to avoid repeating that structure verbatim, we address here just the pure lambda-calculus, unadorned with booleans or numbers). However, the operation of substituting a term for a variable involves some surprising subtleties.

9. Strictly speaking, as we defined it, `iszro t` evaluates to a *representation of* `true` as another term, but let's elide that distinction to simplify the present discussion. An analogous story can be given to explain in what sense the Church booleans represent the real ones.

Syntax

As in Chapter 3, the abstract grammar defining terms (on page 53) should be read as shorthand for an inductively defined set of abstract syntax trees.

5.3.1 Definition [Terms]: Let $\mathcal{V}$ be a countable set of variable names. The set of terms is the smallest set $\mathcal{T}$ such that

1. $\mathtt{x} \in \mathcal{T}$ for every $\mathtt{x} \in \mathcal{V}$;
2. if $\mathtt{t}_1 \in \mathcal{T}$ and $\mathtt{x} \in \mathcal{V}$, then $\lambda\mathtt{x}.\mathtt{t}_1 \in \mathcal{T}$;
3. if $\mathtt{t}_1 \in \mathcal{T}$ and $\mathtt{t}_2 \in \mathcal{T}$, then $\mathtt{t}_1\ \mathtt{t}_2 \in \mathcal{T}$. □

The *size* of a term t can be defined exactly as we did for arithmetic expressions in Definition 3.3.2. More interestingly, we can give a simple inductive definition of the set of variables appearing free in a lambda-term.

5.3.2 Definition: The set of *free variables* of a term t, written $FV(\mathtt{t})$, is defined as follows:

$$\begin{array}{lcl} FV(\mathtt{x}) & = & \{\mathtt{x}\} \\ FV(\lambda\mathtt{x}.\mathtt{t}_1) & = & FV(\mathtt{t}_1) \setminus \{\mathtt{x}\} \\ FV(\mathtt{t}_1\ \mathtt{t}_2) & = & FV(\mathtt{t}_1) \cup FV(\mathtt{t}_2) \end{array}$$ □

5.3.3 Exercise [★★]: Give a careful proof that $|FV(\mathtt{t})| \leq size(\mathtt{t})$ for every term t. □

Substitution

The operation of substitution turns out to be quite tricky, when examined in detail. In this book, we will actually use two different definitions, each optimized for a different purpose. The first, introduced in this section, is compact and intuitive, and works well for examples and in mathematical definitions and proofs. The second, developed in Chapter 6, is notationally heavier, depending on an alternative "de Bruijn presentation" of terms in which named variables are replaced by numeric indices, but is more convenient for the concrete ML implementations discussed in later chapters.

It is instructive to arrive at a definition of substitution via a couple of wrong attempts. First, let's try the most naive possible recursive definition. (Formally, we are defining a function $[\mathtt{x} \mapsto \mathtt{s}]$ by induction over its argument t.)

$$\begin{array}{lcll} [\mathtt{x} \mapsto \mathtt{s}]\mathtt{x} & = & \mathtt{s} & \\ [\mathtt{x} \mapsto \mathtt{s}]\mathtt{y} & = & \mathtt{y} & \text{if } \mathtt{x} \neq \mathtt{y} \\ [\mathtt{x} \mapsto \mathtt{s}](\lambda\mathtt{y}.\mathtt{t}_1) & = & \lambda\mathtt{y}.\ [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1 & \\ [\mathtt{x} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) & = & ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1)\ ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2) & \end{array}$$

This definition works fine for most examples. For instance, it gives

$$[\mathtt{x} \mapsto (\lambda\mathtt{z.\ z\ w})](\lambda\mathtt{y.x}) = \lambda\mathtt{y.}\lambda\mathtt{z.\ z\ w},$$

which matches our intuitions about how substitution should behave. However, if we are unlucky with our choice of bound variable names, the definition breaks down. For example:

$$[\mathtt{x} \mapsto \mathtt{y}](\lambda\mathtt{x.x}) = \lambda\mathtt{x.y}$$

This conflicts with the basic intuition about functional abstractions that *the names of bound variables do not matter*—the identity function is exactly the same whether we write it λ`x.x` or λ`y.y` or λ`franz.franz`. If these do not behave exactly the same under substitution, then they will not behave the same under reduction either, which seems wrong.

Clearly, the first mistake that we've made in the naive definition of substitution is that we have not distinguished between *free* occurrences of a variable `x` in a term `t` (which should get replaced during substitution) and *bound* ones, which should not. When we reach an abstraction binding the name `x` inside of `t`, the substitution operation should stop. This leads to the next attempt:

$$\begin{array}{lcll}
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{x} & = & \mathtt{s} & \\
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{y} & = & \mathtt{y} & \text{if } \mathtt{y} \neq \mathtt{x} \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda\mathtt{y.t}_1) & = & \begin{cases} \lambda\mathtt{y.\ t}_1 & \text{if } \mathtt{y} = \mathtt{x} \\ \lambda\mathtt{y.\ }[\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1 & \text{if } \mathtt{y} \neq \mathtt{x} \end{cases} & \\
[\mathtt{x} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) & = & ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1)\ ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2) &
\end{array}$$

This is better, but still not quite right. For example, consider what happens when we substitute the term `z` for the variable `x` in the term λ`z.x`:

$$[\mathtt{x} \mapsto \mathtt{z}](\lambda\mathtt{z.x}) = \lambda\mathtt{z.z}$$

This time, we have made essentially the opposite mistake: we've turned the constant function λ`z.x` into the identity function! Again, this occurred only because we happened to choose `z` as the name of the bound variable in the constant function, so something is clearly still wrong.

This phenomenon of free variables in a term `s` becoming bound when `s` is naively substituted into a term `t` is called *variable capture*. To avoid it, we need to make sure that the bound variable names of `t` are kept distinct from the free variable names of `s`. A substitution operation that does this correctly is called *capture-avoiding substitution*. (This is almost always what is meant by the unqualified term "substitution.") We can achieve the desired effect by adding another side condition to the second clause of the abstraction case:

$$
\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda y.t_1) & = & \begin{cases} \lambda y.\ t_1 & \\ \lambda y.\ [x \mapsto s]t_1 & \end{cases} & \begin{array}{l}\text{if } y = x \\ \text{if } y \neq x \text{ and } y \notin FV(s)\end{array} \\
[x \mapsto s](t_1\ t_2) & = & ([x \mapsto s]t_1\ ([x \mapsto s]t_2) &
\end{array}
$$

Now we are almost there: this definition of substitution does the right thing *when it does anything at all*. The problem now is that our last fix has changed substitution into a partial operation. For example, the new definition does not give any result at all for $[x \mapsto y\ z](\lambda y.\ x\ y)$: the bound variable y of the term being substituted into is not equal to x, but it does appear free in $(y\ z)$, so none of the clauses of the definition apply.

One common fix for this last problem in the type systems and lambda-calculus literature is to work with terms "up to renaming of bound variables." (Church used the term *alpha-conversion* for the operation of consistently renaming a bound variable in a term. This terminology is still common—we could just as well say that we are working with terms "up to alpha-conversion.")

5.3.4 Convention: Terms that differ only in the names of bound variables are interchangeable in all contexts. □

What this means in practice is that the name of any λ-bound variable can be changed to another name (consistently making the same change in the body of the λ), at any point where this is convenient. For example, if we want to calculate $[x \mapsto y\ z](\lambda y.\ x\ y)$, we first rewrite $(\lambda y.\ x\ y)$ as, say, $(\lambda w.\ x\ w)$. We then calculate $[x \mapsto y\ z](\lambda w.\ x\ w)$, giving $(\lambda w.\ y\ z\ w)$.

This convention renders the substitution operation "as good as total," since whenever we find ourselves about to apply it to arguments for which it is undefined, we can rename as necessary, so that the side conditions are satisfied. Indeed, having adopted this convention, we can formulate the definition of substitution a little more tersely. The first clause for abstractions can be dropped, since we can always assume (renaming if necessary) that the bound variable y is different from both x and the free variables of s. This yields the final form of the definition.

5.3.5 Definition [Substitution]:

$$
\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda y.t_1) & = & \lambda y.\ [x \mapsto s]t_1 & \text{if } y \neq x \text{ and } y \notin FV(s) \\
[x \mapsto s](t_1\ t_2) & = & [x \mapsto s]t_1\ [x \mapsto s]t_2 &
\end{array}
$$
□