Alonzo Church
inventor of lambda calculus

9 Lambda

Let's say we want to add three to each of the numbers in a sentence. Using the tools from Chapter 8, we would do it like this:

```
(define (add-three number)
  (+ number 3))

(define (add-three-to-each sent)
  (every add-three sent))

> (add-three-to-each '(1 9 9 2))
(4 12 12 5)
```

It's slightly annoying to have to define a helper procedure `add-three` just so we can use it as the argument to `every`. We're never going to use that procedure again, but we still have to come up with a name for it. We'd like a general way to say "here's the function I want you to use" without having to give the procedure a name. In other words, we want a general-purpose procedure-generating procedure!

`Lambda` is the name of a special form that generates procedures. It takes some information about the function you want to create as arguments and it returns the procedure. It'll be easier to explain the details after you see an example.

```
(define (add-three-to-each sent)
  (every (lambda (number) (+ number 3)) sent))

> (add-three-to-each '(1 9 9 2))
(4 12 12 5)
```

The first argument to `every` is, in effect, the same procedure as the one we called `add-three` earlier, but now we can use it without giving it a name. (Don't make the mistake of thinking that `lambda` is the argument to `every`. The argument is *the procedure returned by* `lambda`.)

Perhaps you're wondering whether "lambda" spells something backward. Actually, it's the name of the Greek letter L, which looks like this: λ. It would probably be more sensible if `lambda` were named something like `make-procedure`, but the name `lambda` is traditional.*

Creating a procedure by using `lambda` is very much like creating one with `define`, as we've done up to this point, except that we don't specify a name. When we create a procedure with `define`, we have to indicate the procedure's name, the names of its arguments (i.e., the formal parameters), and the expression that it computes (its body). With `lambda` we still provide the last two of these three components.

As we said, `lambda` is a special form. This means, as you remember, that its arguments are not evaluated when you invoke it. The first argument is a sentence containing the formal parameters; the second argument is the body. What `lambda` returns is an unnamed procedure. You can invoke that procedure:

```
> ((lambda (a b) (+ (* 2 a) b)) 5 6)
16

> ((lambda (wd) (word (last wd) (first wd))) 'impish)
HI
```

In real life, though, you're not likely to create a procedure with `lambda` merely to invoke it once. More often, we use `lambda` as in the first example in this chapter, to provide a procedure as argument to a higher-order function. Here are some more examples:

```
> (every (lambda (wd) (se (first wd) wd (last wd)))
         '(only a northern song))
(O ONLY Y A A A N NORTHERN N S SONG G)
```

* It comes from a branch of mathematical logic called "lambda calculus" that's about the formal properties of functions. The inclusion of first-class functions in Lisp was inspired by this mathematical work, so Lisp borrowed the name `lambda`.

```
> (keep (lambda (n) (member? 9 n)) '(4 81 909 781 1969 1776))
(909 1969)

> (accumulate (lambda (this that)
                (if (> (count this) (count that)) this that))
              '(wild honey pie))
HONEY

> (keep (lambda (person) (member? person '(john paul george ringo)))
        '(mick smokey paul diana bill geddy john yoko keith reparata))
(PAUL JOHN)

> (keep (lambda (person) (member? 'e person))
        '(mick smokey paul diana bill geddy john yoko keith reparata))
(SMOKEY GEDDY KEITH REPARATA)
```

Procedures That Return Procedures

An even more powerful use of `lambda` is to provide the value returned by some procedure that you write. Here's the classic example:

```
(define (make-adder num)
  (lambda (x) (+ x num)))

> ((make-adder 4) 7)
11

> (every (make-adder 6) '(2 4 8))
(8 10 14)
```

The value of the expression `(make-adder 4)` is a *procedure,* not a number. That unnamed procedure is the one that adds 4 to its argument. We can understand this by applying the substitution model to `make-adder`. We substitute `4` for `num` in the body of `make-adder`; we end up with

```
(lambda (x) (+ x 4))
```

and then we evaluate that expression to get the desired procedure.

Here's a procedure whose argument is a procedure:

```
(define (same-arg-twice fn)
  (lambda (arg) (fn arg arg)))
```

```
> ((same-arg-twice word) 'hello)
HELLOHELLO

> ((same-arg-twice *) 4)
16
```

When we evaluate `(same-arg-twice word)` we substitute the procedure `word` for the formal parameter `fn`, and the result is

```
(lambda (arg) (word arg arg))
```

One more example:

```
(define (flip fn)
  (lambda (a b) (fn b a)))

> ((flip -) 5 8)
3

> ((flip se) 'goodbye 'hello)
(HELLO GOODBYE)
```

The Truth about `Define`

Remember how we said that creating a procedure with `lambda` was a lot like creating a procedure with `define`? That's because the notation we've been using with `define` is an abbreviation that combines two activities: creating a procedure and giving a name to something.

As you saw in Chapter 7, `define`'s real job is to give a name to some value:

```
> (define pi 3.141592654)

> (* pi 10)
31.41592654

> (define drummer '(ringo starr))

> (first drummer)
RINGO
```

When we say

```
(define (square x) (* x x))
```

it's actually an abbreviation for

```
(define square (lambda (x) (* x x)))
```

In this example, the job of `lambda` is to create a procedure that multiplies its argument by itself; the job of `define` is to name that procedure `square`.

In the past, without quite saying so, we've talked as if the name of a procedure were understood differently from other names in a program. In thinking about an expression such as

```
(* x x)
```

we've talked about substituting some actual value for the `x` but took the `*` for granted as meaning the multiplication function.

The truth is that we have to substitute a value for the `*` just as we do for the `x`. It just happens that `*` has been predefined to have the multiplication procedure as its value. This definition of `*` is *global,* like the definition of `pi` above. "Global" means that it's not a formal parameter of a procedure, like `x` in `square`, but has a permanent value established by `define`.

When an expression is evaluated, every name in the expression must have some value substituted for it. If the name is a formal parameter, then the corresponding actual argument value is substituted. Otherwise, the name had better have a global definition, and *that* value is substituted. It just so happens that Scheme has predefined a zillion names before you start working, and most of those are names of primitive procedures.

(By the way, this explains why when you make a typing mistake in the name of a procedure you might see an error message that refers to variables, such as "variable `frist` not bound." You might expect it to say "`frist` is not a procedure," but the problem is no different from that of any other name that has no associated value.)

Now that we know the whole truth about `define`, we can use it in combination with the function-creating functions in these past two chapters.

```
> (define square (same-arg-twice *))

> (square 7)
49
```

```
> (define fourth-power (repeated square 2))

> (fourth-power 5)
625
```

The Truth about `Let`

In Chapter 7 we introduced `let` as an abbreviation for the situation in which we would otherwise define a helper procedure in order to give names to commonly-used values in a calculation. We started with

```
(define (roots a b c)
  (roots1 a b c (sqrt (- (* b b) (* 4 a c)))))

(define (roots1 a b c discriminant)
  (se (/ (+ (- b) discriminant) (* 2 a))
      (/ (- (- b) discriminant) (* 2 a))))
```

and introduced the new notation

```
(define (roots a b c)
  (let ((discriminant (sqrt (- (* b b) (* 4 a c)))))
    (se (/ (+ (- b) discriminant) (* 2 a))
        (/ (- (- b) discriminant) (* 2 a)))))
```

to avoid creating an otherwise-useless named procedure. But now that we know about unnamed procedures, we can see that `let` is merely an abbreviation for creating and invoking an anonymous procedure:

```
(define (roots a b c)
  ((lambda (discriminant)
     (se (/ (+ (- b) discriminant) (* 2 a))
         (/ (- (- b) discriminant) (* 2 a))))
   (sqrt (- (* b b) (* 4 a c)))))
```

What's shown in boldface above is the part that invokes the procedure created by the lambda, including the actual argument expression.

Just as the notation to define a procedure with parentheses around its name is an abbreviation for a `define` and a `lambda`, the `let` notation is an abbreviation for a `lambda` and an invocation.

Name Conflicts

When a procedure is created inside another procedure, what happens if you use the same formal parameter name in both?

```
(define (f x)
  (lambda (x) (+ x 3)))
```

Answer: Don't do it.

What actually happens is that the inner `x` wins; that's the one that is substituted into the body. But if you find yourself in this situation, you are almost certainly doing something wrong, such as using nondescriptive names like `x` for your variables.

Named and Unnamed Functions

Although you've been running across the idea of function since high school algebra, you've probably never seen an *unnamed* function until now. The high school function notation, $g(x) = 3x + 8$, requires you to give the function a name (g in this case) when you create it. Most of the functions you know, both in math and in programming, have names, such as logarithm or `first`.*

When do you want to name a function, and when not? It may help to think about an analogy with numbers. Imagine if every Scheme number had to have a name before you could use it. You'd have to say

```
> (define three 3)
> (define four 4)
> (+ three four)
7
```

This is analogous to the way we've dealt with procedures until now, giving each one a name. Sometimes it's much easier to use a number directly, and it's silly to have to give it a name.

But sometimes it isn't silly. A common example that we've seen earlier is

* Professional mathematicians do have a notation for unnamed functions, by the way. They write $(x \mapsto 3x + 8)$.

```
(define pi 3.141592654)

(define (circle-area radius)
  (* pi radius radius))

(define (circumference radius)
  (* 2 pi radius))

(define (sphere-surface-area radius)
  (* 4 pi radius radius))

(define (sphere-volume radius)
  (* (/ 4 3) pi radius radius radius))
```

If we couldn't give a name to the number 3.141592654, then we'd have to type it over and over again. Apart from the extra typing, our programs would be harder to read and understand. Giving π a name makes the procedures more self-documenting. (That is, someone else who reads our procedures will have an easier time understanding what we meant.)

It's the same with procedures. If we're going to use a procedure more than once, and if there's a meaningful name for it that will help clarify the program, then we define the procedure with `define` and give it a name.

```
(define (square x) (* x x))
```

`Square` deserves a name both because we use it often and because there is a good traditional name for it that everyone understands. More important, by giving `square` a name, we are shifting attention from the process by which it works (invoking the multiplication procedure) to its *purpose,* computing the square of a number. From now on we can think about squaring as though it were a Scheme primitive. This idea of naming something and forgetting the details of its implementation is what we've been calling "abstraction."

On the other hand, if we have an unimportant procedure that we're using only once, we might as well create it with `lambda` and without a name.

```
> (every (lambda (x) (last (bl x))) '(all together now))
(L E O)
```

We could have defined this procedure with the name `next-to-last`, but if we're never going to use it again, why bother?

Here's an example in which we use an obscure unnamed function to help us define one that's worth naming:

```
(define (backwards wd) (accumulate (lambda (a b) (word b a)) wd))

> (backwards 'yesterday)
YADRETSEY

> (every backwards '(i saw her standing there))
(I WAS REH GNIDNATS EREHT)
```

Pitfalls

⇒ It's very convenient that `define` has an abbreviated form to define a procedure using a hidden `lambda`, but because there are two notations that differ only subtly—one has an extra set of parentheses—you could use the wrong one by mistake. If you say

```
(define (pi) 3.141592654)
```

you're not defining a variable whose value is a number. Instead the value of `pi` will be a *procedure.* It would then be an error to say

```
(* 2 pi)
```

⇒ When should the body of your procedure be a `lambda` expression? It's easy to go overboard and say "I'm writing a procedure so I guess I need `lambda`" even when the procedure is supposed to return a word.

The secret is to remember the ideas of *domain* and *range* that we talked about in Chapter 2. What is the range of the function you're writing? Should it return a procedure? If so, its body might be a `lambda` expression. (It might instead be an invocation of a higher-order procedure, such as `repeated`, that returns a procedure.) If your procedure doesn't return a procedure, its body won't be a `lambda` expression. (Of course your procedure might still use a `lambda` expression as an argument to some *other* procedure, such as `every`.)

For example, here is a procedure to keep the words of a sentence that contain the letter `h`. The domain of the function is sentences, and its range is also sentences. (That is, it takes a sentence as argument and returns a sentence as its value.)

```
(define (keep-h sent)
  (keep (lambda (wd) (member? 'h wd)) sent))
```

By contrast, here is a function of a letter that returns a *procedure* to keep words containing that letter.

```
(define (keeper letter)
  (lambda (sent)
    (keep (lambda (wd) (member? letter wd)) sent)))
```

The procedure `keeper` has letters as its domain and procedures as its range. The procedure *returned by* `keeper` has sentences as its domain and as its range, just as `keep-h` does. In fact, we can use `keeper` to define `keep-h`:

```
(define keep-h (keeper 'h))
```

⇒ Don't confuse the *creation* of a procedure with the *invocation* of one. `Lambda` creates a procedure. The procedure is invoked in response to an expression whose first subexpression represents that procedure. That is, the first subexpression could be the *name* of the procedure, or it could be a `lambda` expression if you want to create a procedure and invoke it right away:

```
((lambda (x) (+ x 3)) 6)
```

In particular, when you create a procedure, you specify its formal parameters—the *names* for its arguments. When you invoke the procedure, you specify *values* for those arguments. (In this example, the `lambda` expression includes the formal parameter `x`, but the invocation provides the actual argument `6`.)

Boring Exercises

9.1 What will Scheme print? Figure it out yourself before you try it on the computer.

```
> (lambda (x) (+ (* x 3) 4))

> ((lambda (x) (+ (* x 3) 4)) 10)

> (every (lambda (wd) (word (last wd) (bl wd)))
         '(any time at all))
```

```
> ((lambda (x) (+ x 3)) 10 15)
```

9.2 Rewrite the following definitions so as to make the implicit `lambda` explicit.

```
(define (second stuff)
  (first (bf stuff)))

(define (make-adder num)
  (lambda (x) (+ num x)))
```

9.3 What does this procedure do?

```
(define (let-it-be sent)
  (accumulate (lambda (x y) y) sent))
```

Real Exercises

9.4 The following program doesn't work. Why not? Fix it.

```
(define (who sent)
  (every describe '(pete roger john keith)))

(define (describe person)
  (se person sent))
```

It's supposed to work like this:

```
> (who '(sells out))
(pete sells out roger sells out john sells out keith sells out)
```

In each of the following exercises, write the procedure in terms of `lambda` and higher-order functions. Do not use named helper procedures. If you've read Part IV, don't use recursion, either.

9.5 Write `prepend-every`:

```
> (prepend-every 's '(he aid he aid))
(SHE SAID SHE SAID)

> (prepend-every 'anti '(dote pasto gone body))
(ANTIDOTE ANTIPASTO ANTIGONE ANTIBODY)
```

9.6 Write a procedure `sentence-version` that takes a function *F* as its argument and returns a function *G*. *F* should take a single word as argument. *G* should take a sentence as argument and return the sentence formed by applying *F* to each word of that argument.

```
> ((sentence-version first) '(if i fell))
(I I F)

> ((sentence-version square) '(8 2 4 6))
(64 4 16 36)
```

9.7 Write a procedure called `letterwords` that takes as its arguments a letter and a sentence. It returns a sentence containing only those words from the argument sentence that contain the argument letter:

```
> (letterwords 'o '(got to get you into my life))
(GOT TO YOU INTO)
```

9.8 Suppose we're writing a program to play hangman. In this game one player has to guess a secret word chosen by the other player, one letter at a time. You're going to write just one small part of this program: a procedure that takes as arguments the secret word and the letters guessed so far, returning the word in which the guessing progress is displayed by including all the guessed letters along with underscores for the not-yet-guessed ones:

```
> (hang 'potsticker 'etaoi)
_OT_TI__E_
```

Hint: You'll find it helpful to use the following procedure that determines how to display a single letter:

```
(define (hang-letter letter guesses)
  (if (member? letter guesses)
      letter
      '_))
```

9.9 Write a procedure `common-words` that takes two sentences as arguments and returns a sentence containing only those words that appear both in the first sentence and in the second sentence.

9.10 In Chapter 2 we used a function called `appearances` that returns the number of times its first argument appears as a member of its second argument. Implement `appearances`.

9.11 Write a procedure `unabbrev` that takes two sentences as arguments. It should return a sentence that's the same as the first sentence, except that any numbers in the original sentence should be replaced with words from the second sentence. A number `2` in the first sentence should be replaced with the second word of the second sentence, a `6` with the sixth word, and so on.

```
> (unabbrev '(john 1 wayne fred 4) '(bill hank kermit joey))
(JOHN BILL WAYNE FRED JOEY)

> (unabbrev '(i 3 4 tell 2) '(do you want to know a secret?))
(I WANT TO TELL YOU)
```

9.12 Write a procedure `first-last` whose argument will be a sentence. It should return a sentence containing only those words in the argument sentence whose first and last letters are the same:

```
> (first-last '(california ohio nebraska alabama alaska massachusetts))
(OHIO ALABAMA ALASKA)
```

9.13 Write a procedure `compose` that takes two functions f and g as arguments. It should return a new function, the composition of its input functions, which computes $f(g(x))$ when passed the argument x.

```
> ((compose sqrt abs) -25)
5

> (define second (compose first bf))

> (second '(higher order function))
ORDER
```

9.14 Write a procedure `substitute` that takes three arguments, two words and a sentence. It should return a version of the sentence, but with every instance of the second word replaced with the first word:

```
> (substitute 'maybe 'yeah '(she loves you yeah yeah yeah))
(SHE LOVES YOU MAYBE MAYBE MAYBE)
```

9.15 Many functions are applicable only to arguments in a certain domain and result in error messages if given arguments outside that domain. For example, `sqrt` may require a nonnegative argument in a version of Scheme that doesn't include complex numbers. (In *any* version of Scheme, `sqrt` will complain if its argument isn't a number at all!) Once a program gets an error message, it's impossible for that program to continue the computation.

Write a procedure `type-check` that takes as arguments a one-argument procedure `f` and a one-argument predicate procedure `pred`. `Type-check` should return a one-argument procedure that first applies `pred` to its argument; if that result is true, the procedure should return the value computed by applying `f` to the argument; if `pred` returns false, the new procedure should also return `#f`:

```
> (define safe-sqrt (type-check sqrt number?))

> (safe-sqrt 16)
4

> (safe-sqrt 'sarsaparilla)
#F
```

9.16 In the language APL, most arithmetic functions can be applied either to a number, with the usual result, or to a *vector*—the APL name for a sentence of numbers—in which case the result is a new vector in which each element is the result of applying the function to the corresponding element of the argument. For example, the function `sqrt` applied to `16` returns `4` as in Scheme, but `sqrt` can also be applied to a sentence such as `(16 49)` and it returns `(4 7)`.

Write a procedure `aplize` that takes as its argument a one-argument procedure whose domain is numbers or words. It should return an APLized procedure that also accepts sentences:

```
> (define apl-sqrt (aplize sqrt))

> (apl-sqrt 36)
6

> (apl-sqrt '(1 100 25 16))
(1 10 5 4)
```

9.17 Write `keep` in terms of `every` and `accumulate`.

Project: Scoring Bridge Hands

At the beginning of a game of bridge, each player assigns a value to his or her hand by counting *points*. Bridge players use these points in the first part of the game, the "bidding," to decide how high to bid. (A bid is a promise about how well you'll do in the rest of the game. If you succeed in meeting your bid you win, and if you don't meet the bid, you lose.) For example, if you have fewer than six points, you generally don't bid anything at all.

You're going to write a computer program to look at a bridge hand and decide how many points it's worth. You won't have to know anything about the rest of the game; we'll tell you the rules for counting points.

A bridge hand contains thirteen cards. Each ace in the hand is worth four points, each king is worth three points, each queen two points, and each jack one. The other cards, twos through tens, have no point value. So if your hand has two aces, a king, two jacks, and eight other cards, it's worth thirteen points.

A bridge hand might also have some "distribution" points, which are points having to do with the distribution of the thirteen cards among the four suits. If your hand has only two cards of a particular suit, then it is worth an extra point. If it has a "singleton," only one card of a particular suit, that's worth two extra points. A "void," no cards in a particular suit, is worth three points.

In our program, we'll represent a card by a word like `h5` (five of hearts) or `dk` (king of diamonds).* A hand will be a sentence of cards, like this:

* Why not `5h`? Scheme words that begin with a digit but aren't numbers have to be surrounded with double-quote marks. Putting the suit first avoids that.

```
(sa s10 s7 s6 s2 hq hj h9 ck c4 dk d9 d3)
```

This hand is worth 14 points: ace of spades (4), plus queen of hearts (2), plus jack of hearts (1), plus king of clubs (3), plus king of diamonds (3), plus one more for having only two clubs.

To find the suit of a card, we take its `first`, and to find the rank, we take the `butfirst`. (Why not the `last`?)

We have a particular program structure in mind. We'll describe all of the procedures you need to write; if you turn each description into a working procedure, then you should have a complete program. In writing each procedure, take advantage of the ones you've already written. Our descriptions are ordered *bottom-up*, which means that for each procedure you will already have written the helper procedures you need. (This ordering will help you write the project, but it means that we're beginning with small details. If we were describing a project to help you understand its structure, we'd do it in *top-down* order, starting with the most general procedures. We'll do that in the next chapter, in which we present a tic-tac-toe program as a larger Scheme programming example.)

`Card-val`

Write a procedure `card-val` that takes a single card as its argument and returns the value of that card.

```
> (card-val 'cq)
2

> (card-val 's7)
0

> (card-val 'ha)
4
```

`High-card-points`

Write a procedure `high-card-points` that takes a hand as its argument and returns the total number of points from high cards in the hand. (This procedure does *not* count distribution points.)

```
> (high-card-points '(sa s10 hq ck c4))
9

> (high-card-points '(sa s10 s7 s6 s2 hq hj h9 ck c4 dk d9 d3))
13
```

Count-suit

Write a procedure `count-suit` that takes a suit and a hand as arguments and returns the number of cards in the hand with the given suit.

```
> (count-suit 's '(sa s10 hq ck c4))
2

> (count-suit 'c '(sa s10 s7 s6 s2 hq hj h9 ck c4 dk d9 d3))
2

> (count-suit 'd '(h3 d7 sk s3 c10 dq d8 s9 s4 d10 c7 d4 s2))
5
```

Suit-counts

Write a procedure `suit-counts` that takes a hand as its argument and returns a sentence containing the number of spades, the number of hearts, the number of clubs, and the number of diamonds in the hand.

```
> (suit-counts '(sa s10 hq ck c4))
(2 1 2 0)

> (suit-counts '(sa s10 s7 s6 s2 hq hj h9 ck c4 dk d9 d3))
(5 3 2 3)

> (suit-counts '(h3 d7 sk s3 c10 dq d8 s9 s4 d10 c7 d4 s2))
(5 1 2 5)
```

Suit-dist-points

Write `suit-dist-points` that takes a number as its argument, interpreting it as the

number of cards in a suit. The procedure should return the number of distribution points your hand gets for having that number of cards in a particular suit.

```
> (suit-dist-points 2)
1

> (suit-dist-points 7)
0

> (suit-dist-points 0)
3
```

Hand-dist-points

Write `hand-dist-points`, which takes a hand as its argument and returns the number of distribution points the hand is worth.

```
> (hand-dist-points '(sa s10 s7 s6 s2 hq hj h9 ck c4 dk d9 d3))
1

> (hand-dist-points '(h3 d7 sk s3 c10 dq d8 s9 s4 d10 c7 d4 s2))
3
```

Bridge-val

Write a procedure `bridge-val` that takes a hand as its argument and returns the total number of points that the hand is worth.

```
> (bridge-val '(sa s10 s7 s6 s2 hq hj h9 ck c4 dk d9 d3))
14

> (bridge-val '(h3 d7 sk s3 c10 dq d8 s9 s4 d10 c7 d4 s2))
8
```