

Lambda Calculus Languages

Thomas W. Christopher
4/8/97

Alpha edition. Release date: 3/22/97

Copyright © 1997. Thomas W. Christopher.

You may reproduce this document in its entirety for personal use with the Lambda-1 and Lambda-2 implementations. For educational use at a nonprofit institution, you may reproduce this document for the students provided you inform the author of the course name and number, the institution name and address, and provide electronic links (instructor's e-mail and course home page URL) to be posted on the web. Send the listing to the author at the address or URL given below.

Any other uses of this document, such as incorporation in a derived work or a compilation, require written permission.

The Lambda-1 and Lambda-2 implementations are public domain. Since it is in the public domain, it may be copied and used without restriction. The author makes no warranties of any kind as to the correctness of the Lambda-1 and Lambda-2 implementations or their suitability for any application. The responsibility for the use of the program lies entirely with the user.

To contact the author/publisher

Thomas Christopher
Department of Computer Science and Applied Mathematics
Illinois Institute of Technology
IIT Center
Chicago IL 60616 USA

tc@charlie.cns.iit.edu
<http://www.iit.edu/~tc>

To obtain a up-to-date copy of this document, the Lambda-1 and Lambda-2 implementations, and the TCLL1 parser generator

<http://www.iit.edu/~tc/toolsfor.htm>

Printed in the United States of America

Table of Contents

Chapter 1 Getting Started	1
1.1 Purpose	1
1.2 Installation	1
Chapter 2 Lambda-1	2
2.1 Quick introduction to Lambda-1.....	2
2.2 Syntax	7
2.3 Environments	7
2.4 Data types	7
2.5 Built-in constants and functions	8
Chapter 3 Lambda-2	10
3.1 Differences between Lambda-2 and Lambda-1.....	10
3.2 Syntax	11
3.3 Built-in constants and functions	12
Chapter 4 Elementary programming techniques	13
4.1 Short-circuited Booleans.....	13
4.2 Loops become tail-end recursion	13
4.3 Pitfalls of lazy evaluation	14
Chapter 5 Programming with lists	16
5.1 Lists per se	16

5.2 Lists representing records 17

Chapter 6 Implementing objects with functions 19

6.1 Binary search tree object. 19

6.1.1 Methods of empty trees. 20

Chapter 7 Programming with streams 23

Chapter 1 Getting Started

1.1 Purpose

This is a manual on Lambda-1 and Lambda-2, two variants of a Lambda Calculus-based functional programming language. It is intended to give you enough information to write programs in the two languages.

1.2 Installation

First you will need to acquire and install two other software packages

- 1 Install the Icon programming language available from <http://cs.arizona.edu/icon/www/>.
- 2 Install either the TCLL1 or the TCLLk parser generator available from <http://www.iit.edu/~tc/toolsfor.htm>.

To install the systems under Windows, Windows 95, Windows NT, or DOS.

- 1 put *lambda.zip* in a directory and unzip it
- 2 execute *build1* to install Lambda-1
- 3 execute *build2* to install Lambda-2
- 4 execute *iconx lambda1* to execute Lambda-1
- 5 execute *iconx lambda2* to execute Lambda-2

Under UNIX, you may have remove CNTL-M's at the ends of the lines if you use the Windows ZIP file. The batch files *build1.bat* and *build2.bat* should also work under UNI if you make them executable (i.e. `chmod +x build2.bat`) and use the entire name as a command (i.e. *build2.bat*).

Chapter 2 Lambda-1

The Lambda-1 and Lambda-2 languages are based on the lambda calculus. They are interactive, side-effect-free and statically scoped with curried functions and lazy parameter passing.

2.1 Quick introduction to Lambda-1

Lambda-1 is an interactive system. You type expressions and definitions in to it and it responds. Your commands to the systems are terminated with semicolons.

The most trivial expressions you can type in are constants. Here are the forms of string and numeric constants, shown as typed in and with the system's response:

```
"Hello, world. ";
Hello, world.
1;
1
1.5;
1.5
1.5e-2;
0.015
1.e-2;
0.01
1e-2;
0.01
1.5e+2;
150.0
1.5e1;
15.0
```

Functions are given in prefix form, the function first, followed by the operands. Operators are just functions. Notice that unlike most programming languages, you do not put parentheses around the argument list nor commas between the arguments.

```
+ 1 2;
3
```

If you type in the function name itself, the system will tell you it is a function.

```
+;
a function
```

If you apply the function to too few arguments, the result is also a function.

```
+ 1;
a function
```

This is called “currying,” named after the mathematician Curry. A function of $n > 1$ arguments is really a function of one argument that returns a function of $n-1$ arguments. Operator `+` is a function of one numeric argument that returns a function of one numeric argument that returns a numeric result.

You can define a name to have a value. Then when you use that name, you will get its assigned value.

```
def two=2;
two;
2
```

The value assigned to a name can, of course, be a function. Here we define *inc* to be a function that returns one plus the value of its operand.

```
def inc=+ 1;
inc two;
3
```

Boolean values are assigned to the predefined names *true* and *false*.

```
true;
true
false;
false
```

The usual relational operators are provided and return Boolean values.

```
< 1 2;
true
< 2 1;
false
```

The equal operator, and its complement, test for identity (like the `===` in Icon). An integer will never be considered equal to a real, because they are of different types.

```
= 2 two;
true
= 2.0 two;
false
~= 2 two;
false
~= 2.0 two;
true
```

The *if* function is like the if-then-else statement. It takes three operands. It evaluates the first operand, and if that is true, it returns the second operand. Otherwise it returns the third. While the first operand is always evaluated, only one of the second or third operands is. In the jargon of functional programming, *if* is *strict* in its first operand and *non-strict* in its second and third.

Lambda Calculus

```
if (< 1 2) "true" "false";
true
if (< 3 2) "true" "false";
false
```

The *if* function is curried, just as all the other functions are, so you can apply it to a partial argument list to construct other functions.

```
def first=if true;
def second=if false;
first 1 2;
1
second 1 2;
2
```

The only built-in data structuring operation in Lambda-1 is the *pair*. A pair is constructed with the function *cons*.

```
cons 1 2;
1:2
```

Notice that when the pair is written out, the two components are separated by a colon. The colon associates to the right, so another pair on the right side will not be parenthesized, but one on the left will.

```
cons 1 (cons 2 3);
1:2:3
cons (cons 1 2) 3;
(1:2):3
```

The left and right components of a pair are referred to as the head and tail of the pair. They may be selected by the functions *hd* and *tl*, respectively.

```
def c=cons 1 2;
c;
1:2
hd c;
1
tl c;
2
```

A list is a sequence of elements built out of pairs. The head of a pair is an element of the list; the tail of the pair is the rest of the list. The end of a list is represented by the special value, *nil*.

```
nil;
nil
cons 1 (cons 2 (cons 3 nil));
1:2:3:nil
```

You can test a value to see if it is a pair or is *nil* with the functions *pair?* and *null?*.

```
pair? c;
true
null? c;
false
```

```

null? 1;
false
null? nil;
true
pair? 1;
false

```

Lambda-1 is based on the lambda calculus. In the lambda calculus, functions are defined using the form

$$\lambda x . e$$

where x is the name of the parameter and e is the expression that may refer to parameter x . Since most of us don't have a lambda on our keyboards, Lambda-1 substitutes the back slash for it.

```

def dec=\x.- x 1;
dec 3;
2

```

A function of more than one argument is written, in curried fashion, as a function of one argument that returns another function. For example, the function *fromto* is a function of two arguments, i and n , that will yield a list of all the numbers from i to n .

```

def fromto=\i.\n.if (<= i n) (cons i (fromto (+ i 1) n)) nil;
fromto 1 5;
1:2:3:4:5:nil

```

The system limits the amount of a list it will write out. If the list is longer than the permitted length, it just puts a "....." at the end of the part it writes.

```

def iota=fromto 1;
iota 6;
1:2:3:4:5:6:.....

```

The function *iota* is named after an operator in the language APL which produces an array initialized to the numbers from one through the length of the array¹.

The definitions can be in any order and can refer to each other. For example, the functions *odds* and *evens* yield the odd and even elements of a list. They can be implemented as mutually recursive:

```

def odds=\l.if (pair? l) (cons (hd l) (evens (tl l))) nil;
def evens=\l.if (pair? l) (odds (tl l)) nil;
iota 7;
1:2:3:4:5:6:.....
odds (iota 7);
1:3:5:7:nil
evens (iota 7);
2:4:6:nil

```

¹Or zero through the length minus one, if the APL system is set to origin zero.

Lambda Calculus

One of the most important functions for manipulating lists is *map*, which takes a function and a list and gives you a new list equal to the function applied to every element of the input list.

```
def map=\f.\l.if (pair? l) (cons (f (hd l))(map f (tl l))) l;
map inc (iota 4);
2:3:4:5:nil
map (< 3) (iota 6);
false:false:false:true:true:true:....
map iota (iota 3);
(1:nil):(1:2:nil):(1:2:3:nil):nil
```

Function *cons* is non-strict in its two arguments. In fact, it is *lazy*. It does not evaluate the arguments. The arguments are only evaluated later if and when their values are needed by *hd*, *tl*, or the printing routine. Here, although *iota* has been called with a length of 10000, the system only generates the first six elements of the list.

```
map iota (iota 10000);
(1:nil):(1:2:nil):(1:2:3:nil):(1:2:3:4:nil):(1:2:3:4:5:nil):(1:
2:3:4:5:6:....):....
```

Lambda-1 uses *static scoping*. The meanings of identifiers is determined from the context in which they are bound (created), not the context in which they are subsequently used. For example, suppose we write a function *scaleList* which is to multiply every element of a list by a constant factor.

```
def scaleList=\l.\f.map (\x.* x f) l;
scaleList (iota 10) 5;
5:10:15:20:25:30:....
```

Notice that *scaleList* uses *f* as the name of the factor and calls *map*, passing it function $(\lambda x. * x f)$, to perform the actual operation.

In $(\lambda x. * x f)$, variable *x* is *bound* and *f* is used *free*. When *map* applies function $(\lambda x. * x f)$, it will have to look up the meaning of the free variable *f*. So where does it find the value of *f*? Clearly what we want is for *f* to have the value of the scale factor *f* in our *scaleList* function. But notice that *map* also has a parameter *f*. We don't want that *f*, but a language might look up the meaning of *f* where it is used and get the wrong value.

In fact, Lambda-1 and Lambda-2 will give us the meaning of *f* we want, the meaning it had when the function it is in was defined, not where it is used. This is called *static scoping*. If they looked up the meaning of *f* where it was used—and many languages do that—it would be called *dynamic scoping*.

You can load a collection of definitions from a file with the *load* command. *Load* takes a string as its operand and loads the file named by the string. Actually, the file is inserted into the input beginning with the line following the current line. To avoid having the input mangled, the *load* expression should be alone on a line.

```
load "defs.l1";
```

Lambda-1 produces a log file of the commands it received during the session and the values it computed. The log file is named "L1Log.tmp". If you create some definitions during an interactive session that you want to use again, you can copy them from the log file.

2.2 Syntax

```

start = { stmt }.
stmt = def definition ";" | expr ";" | ";" .
definition = ident "=" expr .
expr = primary { primary }
      | "\" ident "." expr .
primary = number
         | string
         | ident | "="
         | "(" expr ")" .

```

2.3 Environments

While evaluating expressions, the Lambda languages have to look up the meanings of names. For this they use two environments: the global environment and the local environment. An environment is a table mapping variable names to values.

The global environment has all the names of the built-in constants and functions already defined in it. Other names are bound to expressions in the global environment by the `def` command. A new `def` for the same name will replace the previous definition.

The local environment binds of parameter names to their values within function applications.

When the interpreter is trying to look up a variable, it first looks in the local environment and then in the global environment.

2.4 Data types

The data types in the Lambda languages are shown in Table 1.

Table 1 Types in the Lambda languages

Type	Comments
numeric	integer or real
string	character string

Lambda Calculus

Table 1 Types in the Lambda languages

Type	Comments
pair	pair of values, accessed by functions <i>hd</i> and <i>tl</i> , used in constructing lists
nil	special value used to mark the end of a list
environment	a table mapping variable names to values. An environment is an internal data type, not directly visible to the user.
closure	a function bound to an environment
suspension	an unevaluated expression bound to an environment. A suspension is an internal data type, not directly visible to the user.

2.5 Built-in constants and functions

The constants defined in the global environment in Lambda-1 are shown in Table 2. The built-in functions are shown in Table 3.

Table 2 Built-in constants.

Name	Means
true	true, a Boolean value
false	false, a Boolean value
nil	used as an end-of-list marker.

Table 3 Built-in operators.

Application	Means
+ x y	x+y
- x y	x-y
* x y	x*y
/ x y	x/y
mod x y	x mod y, remainder of x divided by y
= x y	x=y, true if x and y are identical, i.e. equal integers, equal floating point numbers, equal strings, or both are the same object.
~= x y	true if x is not identical to y, i.e. if they are not both of the same type or not the same object.

Table 3 Built-in operators.

Application	Means
<code>> x y</code>	$x > y$, the operands must be numeric or strings. If both operands are numeric, a numeric comparison is done, otherwise a string comparison.
<code>>= x y</code>	$x \geq y$, the operands must be numeric or strings. If both operands are numeric, a numeric comparison is done, otherwise a string comparison.
<code>< x y</code>	$x < y$, the operands must be numeric or strings. If both operands are numeric, a numeric comparison is done, otherwise a string comparison.
<code><= x y</code>	$x \leq y$, the operands must be numeric or strings. If both operands are numeric, a numeric comparison is done, otherwise a string comparison.
<code>if x y z</code>	evaluates x . If the value of x is true, it evaluates and yields the value of y . If x is false, it evaluates and yields the value of z . Note that x is always evaluated, but only one of y and z is.
<code>cons x y</code>	constructs a <i>pair</i> whose head is x and whose tail is y . It will be written out as $x:y$. Cons does not evaluate its arguments.
<code>hd x</code>	the head of a pair: $hd (cons x y)$ yields x .
<code>tl x</code>	the tail of a pair: $tl (cons x y)$ yields y .
<code>null? x</code>	true if the value of x is <i>nil</i> .
<code>pair? x</code>	true if the value of x is a pair.
<code>number? x</code>	true if the value of x is a number.
<code>boolean? x</code>	true if the value of x is a Boolean, i.e. true or false.
<code>string? x</code>	true if the value of x is a string.
<code>load x</code>	x must be a string that names a text file containing Lambda-1 definitions and expressions. The lines of the file are inserted in front of the next line of the current input. You should only use this application alone on a line.

Chapter 3 Lambda-2

3.1 Differences between Lambda-2 and Lambda-1.

There are five differences between Lambda-1 and Lambda-2:

- there is new syntax for *if* expression,
- function definitions can use more than one parameter name,
- there is an infix operator for *cons*,
- there is a syntax for local definitions in expressions, and
- there is a syntax for mutually-recursive local definitions in expressions.

There is no function *if* in Lambda-2. The new form is

```
if e1 then e2 else e3
```

which is equivalent to

```
if (e1) (e2) (e3)
```

in Lambda-1.

In Lambda-2, you can list more than one parameter between the "\ " and the ".". The functions are still curried. You can still apply the functions to fewer arguments than they are expecting to get a function that is waiting for the rest.

```
\x y z.e
```

is identical in effect to

```
\x.\y.\z.e
```

The infix colon operator produces a pair. As in the output, the colon operator is right associative. In Lambda-2, the map function can be written

```
def map=\f l.if pair? l then f (hd l): map f (tl l) else l;
```

Notice that if-then-else has lowest precedence, colon has an intermediate precedence, and function application has highest precedence.

The *cons* function still exists. The Lambda-2 system translates the colon operator into *cons*.

In both Lambda-1 and Lambda-2, the parameters to a function are local names, that is, names known within the body of the function. Lambda-2 provides two additional ways to create local names.

The let-expression has the form:

let $n_1 = e_1; n_2 = e_2; \dots; n_k = e_k$ **in** b

where each n_i is a name, e_i is an expression, and b is an expression called the *body* of the let-expression. Each n_i is bound to the corresponding e_i in the body b , but the names n_i are not known within the e_j 's. Each expression e_i can only refer to the names known in the environment surrounding the let-expression. The let-expression is exactly equivalent to

$(\lambda n_1 n_2 \dots n_k = e_k . b) (e_1) (e_2) \dots (e_k)$

If you wish the names to be defined in order and each expression to be able to refer to the previous names, you may use a cascade of let-expressions:

let $n_1 = e_1$ **in let** $n_2 = e_2$ **in let** ... **in let** $n_k = e_k$ **in** b

or use a letrec-expression.

The letrec-expression, or “recursive let-expression”, has the form:

letrec $n_1 = e_1; n_2 = e_2; \dots; n_k = e_k$ **in** b

where each n_i is a name, e_i is an expression, and b is the body of the let-expression. Each n_i is bound to the corresponding e_i in all expressions, e_k , and in the body b . If the expressions are functions, they can call each other recursively.

3.2 Syntax

```

start = stmt { stmt }.
stmt = def definition ";" | expr ";" | ";" .
definition = ident "=" expr .
expr = conses
      | "\" identList "." expr
      | if expr then expr else expr
      | let definition { ";" definition } in expr
      | letrec definition { ";" definition } in expr.
identList = ident { ident }.
conses = application constail.
constail = [ ":" conses ].
application = primary { primary }.
primary = number
          | string
          | ident
          | "="
          | "(" expr ")" .

```

3.3 Built-in constants and functions

The constants and functions built in to Lambda-2 are the same as those in Lambda-1, but there are two differences. The *cons* function is still available, but you will probably use the binary colon operator instead. The *if* function is no longer directly available. You must use the *if-then-else* syntax.

Chapter 4 Elementary programming techniques

4.1 Short-circuited Booleans

As an example of the use of both the new form of if-expression and of the multiparameter functions, consider these definitions of Boolean *and* and *or*:

```
def and=\ e1 e2 . if e1 then e2 else false;
def or=\ e1 e2 . if e1 then true else e2;
```

These are short-circuited boolean operations. Expression *e1* is always evaluated, but *e2* is only evaluated if the value of the expression is not determined by *e1*. This is because, like Lambda-1, Lambda-2 uses lazy evaluation. An expression is not evaluated unless its value is required.

In fact, the only thing that forces any evaluation at all is the attempt of the interpreter to write out an answer.

4.2 Loops become tail-end recursion

Consider the following function in Icon to compute the *n*th Fibonacci number (remember, the first two Fibonacci numbers are 1 and each following Fibonacci numbers is the sum of the two previous):

```
procedure fib(n)
local i,j,t
i:=j:=1
while n>1 do {
    t:=i+j
    i:=j
    j:=t
    n:=n-1
}
return i
end
```

In *fib*, *i* and *j* are two successive Fibonacci numbers. Each time around the loop, *i* becomes *j*, and *j* becomes the sum of *i* and *j*, which is the next pair of successive Fibonacci numbers. When *n* is decremented down to 1, *i* will be the Fibonacci number we are interested in.

Here is how it would be written in Lambda-2. The loop is encoded as the recursive procedure *fib3* which calls itself for each successive iteration.

```
def fib=\n.
  letrec fib3= \n i j.
    if <= n 1 then i
    else fib3 (- n 1) j (+ i j)
  in fib3 n 1 1;
```

4.3 Pitfalls of lazy evaluation

Consider again the above function to write out the n^{th} Fibonacci number. If you try *fib* with too large a value of *n*, you will crash the Icon program interpreting Lambda-2 and get several pages of trace-back. Once most of it finishes scrolling off your screen, you'll see something like the following:

```
eval(record Primitive_1(2,procedure addFn),table_36(4)) from line 150 in xeq2.icn
addFn(list_914 = [record Suspension_10(3)]) from line 109 in xeq2.icn
force(record Suspension_8(record App_8(2),table_34(4),&null)) from line 3 in rts12.icn
eval(record Primitive_1(2,procedure addFn),table_34(4)) from line 150 in xeq2.icn
addFn(list_915 = [record Suspension_6(3)]) from line 109 in xeq2.icn
force(record Suspension_4(record App_8(2),table_32(4),&null)) from line 3 in rts12.icn
```

What has scrolled off the screen is:

```
Run-time error 301
File rts12.icn; Line 3
evaluation stack overflow
```

The recursion has gone too deep.

Strangely, it is not the recursive call of *fib3* in the *else* clause that causes the recursion to go too deep. This *tail-end* recursion is optimized in Lambda-2. What causes the system to crash is the attempt to write out the value *fib* returns.

Whereas *fib3* must evaluate its parameter *n* to decide what to do in the if-expression, it does not evaluate either *i* or *j* because it doesn't need their values. It is only when the output routine needs the value *fib* returned that it forces it to be evaluated. Here's approximately what happens if we call *fib 4*:

```
fib 4 calls fib3 4 1 1
fib3 4 1 1
  evaluates parameter n getting 4
  evaluates <= 4 1
  calls fib3 (- 4 1) 1 (+ 1 1)
fib3 (- 4 1) 1 (+ 1 1)
  evaluates parameter n getting 3
  evaluates <= 3 1
  calls fib3 (-3 1) (+ 1 1) (+ 1(+ 1 1))
fib3 (-3 1) (+ 1 1) (+ 1(+ 1 1))
  evaluates parameter n getting 2
  evaluates <= 2 1
  calls fib3 (- 2 1) (+ 1(+ 1 1)) (+ (+ 1 1)(+ 1(+ 1 1)))
fib3 (- 2 1) (+ 1(+ 1 1)) (+ (+ 1 1)(+ 1(+ 1 1)))
  evaluates parameter n getting 1
  evaluates <= 1 1
  returns parameter i which has the value (+ 1(+ 1 1))
```

To output the value returned, the interpreter evaluates

```
(+ 1(+ 1 1))
```

As you can see, with each recursive call of *fib3* creates larger and larger unevaluated expressions for parameters *i* and *j*. It does not create larger unevaluated

expressions for parameter n because at each call of *fib3*, n gets evaluated and its value is reused.

Chapter 5 Programming with lists

5.1 Lists *per se*

There are only two data-structuring facilities in Lambda-1 and Lambda-2: pairs (also known as *lists* or *cons cells*) and partially parameterized functions (which hold data in their parameter lists). Here we consider some functions that are commonly used on lists.

Recall the map function returns a list of the values from applying a functional parameter to every element of a list.

```
def map=\f l.if pair? l then f (hd l): map f (tl l) else l;
```

We often need to apply a binary function to the corresponding elements of two lists. Here is a function to do that.

```
def map2=\f l m.if pair? l then
  if pair? m then f (hd l) (hd m) : map2 f (tl l) (tl m)
  else m
  else l;
```

We also often need to add up, or multiply, or perform some operation on all the elements of a list. The function reduce does that.

```
def reduce = \f l.
  if pair? (tl l) then f (hd l) (reduce f (tl l))
  else (hd l);
```

Reduce is like putting a *right associative* binary operator between every element of a list. For example,

```
reduce - (1:2:3:4:nil);
-2
```

since $\text{reduce } - (1:2:3:4:\text{nil})$ is equal to $1-(2-(3-4))$, and

```
reduce - (1:2:3:nil);
2
```

Scan is a function of two arguments, a function f and a list l and builds a list of the results of applying the function to initial segments of the list. For example,

```
scan f (a:b:c:d:e:nil)
```

yields

```
a : f a b : f (f a b) c : f ( f (f a b) c) d : f ( f ( f (f a b) c) d) e : nil
```

Here are two examples.

```
scan - (1:2:3:4:5:nil);
1:-1:-4:-8:-13:nil
scan + (1:2:3:4:5:nil);
1:3:6:10:15:nil
```

Here is a definition of scan:

```
def scan = \f l.
  letrec
    scan3=\v f l.
      cons v (if pair? l then
              scan3 (f v (hd l)) f (tl l)
              else nil)
  in if pair? l then scan3 (hd l) f (tl l) else nil;
```

5.2 Lists representing records

Although the Lambda languages do not have records or structures, it is possible to simulate records by short lists whose elements are the fields of the record. For example, suppose we wish to implement binary search trees. Every node in the tree needs four fields: *key*, *value*, *left* subtree pointer, and *right* subtree pointer. We may implement a node as a list:

key:value:left:right

But there remains a greater difficulty. Most algorithms written in terms of records and pointers require that we be able to assign new values to the fields of a record. In the Lambda languages, we cannot assign new values. What we have do is create a new data structure with the changes made in it.

In the case of the binary search tree, if we add a node, we must change the node above it in the tree. That means we must copy that node with a new pointer in it. And since we've created a new copy of that node, we must change the node above it, and so on all the way back to the root. However, subtrees that have not changed can be left alone and shared between the old and the new tree.

Here is code for insert. It inserts a key, *k*, and value, *v*, into tree, *t*.

```
def insert=\t k v.
  if null? t then k:v:nil:nil
  else let a=hd t;
        b=hd (tl t);
        c=hd (tl (tl t));
        d=tl (tl (tl t))
  in if < k a then a:b:insert c k v:d
     else if > k a then a:b:c:insert d k v
     else k:v:c:d;
```

If the tree is empty (*null? t*) then we create a new node. Otherwise let the node have the fields *a:b:c:d*, i.e. *a* is the key, *b* the value, *c* the left subtree, and *d* the right subtree. If the key we are inserting is less than *a*, we return a new node identical to the current except that there is a new left subtree which is the result of inserting key and value into subtree *c*. Similarly if *k* is greater than *a*, we re-

Lambda Calculus

turn a new node identical to the current except that the right subtree is different. If k equals a , we replace the node with one that has an equal key, a new value, and the same subtrees.

Looking up values in the tree is trivial:

```
def lookup=\t k.
  if null? t then nil
  else if < k (hd t) then lookup (hd (tl (tl t))) k
  else if > k (hd t) then lookup (tl (tl (tl t))) k
  else hd (tl t);
```

We decided arbitrarily that an absent key will yield the value *nil*

To test these functions, we filled a tree from a list of key:value pairs. Function `fill_bst_list` takes a tree and a key:value list as parameters. It inserts the key:value pairs one at a time into to tree and returns the final resulting tree.

The list we used to fill the tree was called *bstdata*. It tests adding left subtrees, right subtrees and replacing a node.

Here are the results of a test:

```
def fill_bst_list=\t l.
  if null? l then t
  else fill_bst_list (insert t (hd (hd l)) (tl (hd l))) (tl l);

def bstdata=("S":1):("X":2):("A":3):("H":4):("A":5):nil;

def X=fill_bst_list nil bstdata;
lookup X "Z";
nil
lookup X "B";
nil
lookup X "H";
4
```

Chapter 6 Implementing objects with functions

Objects, in the object-oriented programming sense, are capsules of data and procedures. The procedures, typically called *methods*, are called to manipulate the object.

In the Lambda languages, an object is a function, or more precisely, a *closure* which is a function bound in an environment. The names bound in the environment represent the data fields of the object. The function that represents an object we will call the object's *interface function*. The interface function will take the name of the method desired and will return that method's function. If the method requires no additional parameters, the interface function can perform the operation itself, rather than returning a function to do it.

6.1 Binary search tree object

Let us return to the example of the binary search tree from Section 5.2 on page 17. Here we create a definition of a binary search tree object.

Function `BST` creates a new, empty binary search tree. `BST` takes a parameter that represents the default value for keys not in the tree. There are five methods defined for `BST`'s: "type", "size", "insert", "lookup", and "list". The operations on binary search trees are summarized in Table 4.

Table 4 Operations on `BST`, binary search tree, objects.

operation	meaning
<code>BST d</code>	create a new binary search tree with default value <code>d</code> .
<code>t "type"</code>	"BST", if object <code>t</code> is a BST.
<code>t "size"</code>	returns the number of key/value pairs in the tree.
<code>t "insert" k v</code>	return a new BST obtained from <code>t</code> by inserting key <code>k</code> with value <code>v</code> .
<code>t "lookup" k</code>	returns the value associated with key <code>k</code> in BST <code>t</code> . Returns the default value, <code>d</code> , if key <code>k</code> is not present in the table.
<code>t "list" x</code>	returns a list of all the key/value pairs in <code>t</code> in sorted order followed by the list <code>x</code> .

Although Table 4 shows how the methods are used, to understand the code, you must understand that the interface function is really a function of one parameter, the method name. For example, `t "insert"` returns a function of two parameters, `k v`, which actually constructs the new binary search tree.

Here is a definition for BST.

```
def BST = \default.  
  letrec  
    empty=\method.  
      if = method "lookup" then \k.default  
      else if = method "insert" then node empty empty  
      else if = method "list" then \tail.tail  
      else if = method "type" then "BST"  
      else if = method "size" then 0  
      else "illegal method to BST";  
    node=\left right key val.  
      \method.  
        if = method "lookup" then  
          \k. if < k key then left "lookup" k  
              else if > k key then right "lookup" k  
              else val  
        else if = method "insert" then  
          \k v.if < k key then  
            node (left "insert" k v) right key val  
            else if > k key then  
              node left (right "insert" k v) key val  
            else node left right k v  
        else if = method "list" then  
          \tail.left "list" ((key:val):right "list" tail)  
        else if = method "type" then "BST"  
        else if = method "size" then  
          + 1 (+ (left "size") (right "size"))  
        else "illegal method to BST"  
  in empty;
```

The outer function takes a parameter, *default*, the default value for the tree, and returns an empty tree. There are two kinds of tree: an empty tree and a node containing a left subtree, a right subtree, a key, and a value. The two definitions in the `letrec` define the two kinds of tree.

Both *empty* and *node* yield interface functions which take a string parameter, *method*. Both test *method* against the same set of strings. The difference is that *node* takes four parameters: the left subtree, the right subtree, the key and the value. These parameters become part of the surrounding environment for the node's interface function.

Instead of elements of a list, the data fields of the object are just non-local variables.

To see how the BST works, we will examine how each kind of BST (*empty* and *node*) handles each of the methods. In the examples, *e* is an empty BST and *n* is a node.

6.1.1 *Methods of empty trees*

```
e "lookup" k
```

is supposed to look up a key in the tree and return the associated value, but there are no keys in an empty tree, so whatever key is supplied, the value will be the default. So `"lookup"` returns a function, `\k.default`, that ignores its argument, the key, and returns the default value.

```
e "insert" k v
```

is supposed to insert a key/value pair into the tree creating a new tree. Since the tree is empty, this will be the only key/value pair in the new tree. The `"insert"` method uses a trick to create a node: currying. The node function is defined to take the two subtree pointers first, followed by the key and the value. The `"insert"` method returns a closure for node with the subtree parameters already bound to the empty tree. This closure is expecting the key and value parameters next, which are supplied by the `k` and `v` following the `e "insert"`.

```
e "type"
```

asks the object `e` to tell what type object it is. The answer is `"BST"`.

```
e "size"
```

asks the object `e` to tell how many key/value pairs it contains. The answer is `0`.

```
e "list" tail
```

constructs a list of key/value pairs from the object which are to be followed by the list `tail`. For an empty object, the value is simply `tail`. From outside the object, we would normally pass `nil` as the tail, but `tail` is used inside the definition to construct the list.

```
n "type"
```

asks the object `n` to tell what type object it is. For a node as for an empty object, the answer is `"BST"`.

```
n "size"
```

asks the object `n` to tell how many key/value pairs its BST contains. The answer is one plus the number in the left subtree plus the number in the right subtree. This is expressed as

```
+ 1 (+ (left "size") (right "size"))
```

```
n "lookup" k
```

looks up the key, `k`, in the tree and returns its associated value. If the key in this node equals `k`, then the value is returned. Otherwise, the lookup is passed on to the appropriate subtree.

```
n "insert" k v
```

Lambda Calculus

creates a copy of the BST rooted in node `n` with the key `k` and the value `v` inserted. If the key `k` is less than the key at `n`, it creates a new node identical to `n` except that the left subtree has `k` and `v` inserted into it. If the key `k` is greater than the key at `n`, it creates a new node identical to `n` except that the right subtree has `k` and `v` inserted into it. If `k` equals the key, the new node is identical to `n` except that the value is replaced by `v`.

```
n "list" tail
```

creates a list composed of the sorted associations in the left subtree followed by the key/value pair in the current node followed by the sorted associations in the right subtree followed by the list tail. This is actually rather trivial using the `"list"` method recursively:

```
\tail.left "list" ((key:val):right "list" tail)
```

Here is a test of the BST object:

```
def buildBST=\default l.  
  letrec  
    fill=\t l.  
      if null? l then t  
      else fill (t "insert" (hd (hd l)) (tl (hd l))) (tl l)  
    in fill (BST default) l;  
def bstdata=("S":1):("X":2):("A":3):("H":4):("A":5):nil;  
  
def T=buildBST "N/A" bstdata;  
T "list" nil;  
(A:5):(H:4):(S:1):(X:2):nil  
T "size";  
4  
T "lookup" "Z";  
N/A  
T "lookup" "B";  
N/A  
T "lookup" "H";  
4  
T "type";  
BST  
BST nil "type";  
BST  
T "frog";  
illegal method to BST  
BST nil "frog";  
illegal method to BST  
buildBST "none" (T "list" nil) "list" nil;  
(A:5):(H:4):(S:1):(X:2):nil
```

Chapter 7 Programming with streams

Since the Lambda languages use lazy evaluation, it is possible to define lists that are conceptually infinite in length. But since only a finite portion is accessed, these lists require only a finite amount of storage and time to construct. Some algorithms can be expressed conveniently in terms of such lists.

One term for such an infinite list is *stream*, although that term has other meanings as well.

Here is a stream of the number 1 repeated boundlessly many times:

```
def ones=1:ones;
ones;
1:1:1:1:1:1:1:1:1:1:1:....
```

Here are two functions that generate infinite lists of numbers. Function *from* generates a list of integers that starts from a given value and steps by one. It is defined in terms of function *byfrom* that takes two arguments, a step size and an initial value. It generates a list whose first element is the initial value and all of whose subsequent elements differ from the previous by the step.

```
def byfrom = \j i. i:byfrom j (+ i j);
def from = byfrom 1;
```

We can define the natural numbers as the integers starting from one:

```
def naturals=from 1;
naturals;
1:2:3:4:5:6:7:8:9:10:11:....
```

Function *take n l* will give a list consisting the initial *n* elements taken from list *l*. Function *drop n l*, on the other hand, gives the remainder of list *l* with the first *n* elements removed.

```
def take = \n l.if > n 0 then hd l:take (- n 1) (tl l) else nil;
def drop = \n l.if > n 0 then drop (- n 1) (tl l) else l;
```

Function *iota n* that gives a list of the integers from one through *n* can be defined in terms of *take* and *from*:

```
def iota = \n. take n (from 1);
```

The Fibonacci numbers (see section 4.2), can be defined as the numbers in this stream:

```
def fibs=1:1:map2 + fibs (tl fibs);
```

Lambda Calculus

```
fibs;  
1:1:2:3:5:8:13:21:34:55:89:....  
tl fibs;  
1:2:3:5:8:13:21:34:55:89:144:....  
map2 + fibs (tl fibs);  
2:3:5:8:13:21:34:55:89:144:233:....
```

Function *map2* is defined in Chapter 5. It applies a function to the corresponding elements taken from two lists. The first two elements of the stream *fibs* are defined to be the number 1. The rest of the stream *fibs* is defined to be *map2 + fibs (tl fibs)*, as can be seen from the example.

The factorial of an integer *n* greater than or equal to one is the product of the integers from 1 through *n*. We can define a stream of factorials using *scan* on the natural numbers:

```
def factorials=scan * naturals;  
factorials;  
1:2:6:24:120:720:5040:40320:362880:3628800:39916800:....
```

If streams *l* and *m* are ascending streams of numbers, function *without l m* will yield the stream of numbers in stream *l* that are not in stream *m*.

```
def without = \l m.let lh=hd l; lt=tl l; mh=hd m; mt=tl m  
  in if < lh mh then lh:without lt m  
     else if > lh mh then without l mt  
     else without lt mt;
```

Using *without*, we can define a stream of primes.

```
def primes = letrec  
  primefilter = \l.  
    letrec s = hd l;  
          v = + s s  
          in s:primefilter (without (tl l) (byfrom s v))  
    in primefilter (from 2);
```

The contained function *primefilter* will take a stream of ascending integers and use *without* to remove from that stream all the multiples of the first element and will apply itself to the tail of the resulting stream. We apply *primefilter* to the stream of natural numbers beginning at 2. Here's how it works:

from 2 yields 2:3:4:5:6:7:8:9:.....

primefilter 2:3:4:5:6:7:8:9:.... puts 2 into the output stream and uses *without* to remove multiples of 2 from the rest of the sequence, yielding 3:5:7:9:11:13:15:....

primefilter 3:5:7:9:11:13:15:... puts 3 into the output stream and uses *without* to remove multiples of 3 from the rest of the sequence, yielding 5:7:11:13:17:19:23:....

and so on.

```
primes;  
2:3:5:7:11:13:17:19:23:29:31:....
```

A composite number is a product of primes. All natural numbers except primes are composites, so we can define composites as follows:

```
def composites=without naturals primes;  
composites;  
1:4:6:8:9:10:12:14:15:16:18:....
```