Chapter 1 Tour of Lambda1

1.1 Files

The files related to Lambda-1 are

- build1.bat—a batch file to build the Lambda-1 compiler/interpreter.
- lambda1.grm—the grammar (in TCLL1 form) for the Lambda-1 parser.
- lambda1.ll1—the parse tables resulting from passing lambda1.grm through TCLL1.
- lambda1.icn—the main program for Lambda-1.
- lamscan1.icn—the scanner for Lambda-1.
- lamsem1.icn—the semantics (action) routines for Lambda-1. These build a tree representation for the Lambda-1 definitions and expressions.
- xeq1.icn—the routines to interpret the tree representations expressions. This file contains the definitions of the data structures used for both the program tree and data structures.
- rts12.icn—the routines to execute the primitive functions used by both Lambda-1 and Lambda-2.
- parsell1.icn, readll1.icn, and semstk.icn—routines from the TCLL1 parser.

1.2 Module Structure

The modules and call structure of Lambda-1 are pictured in Figure 1.

The main program (in lambda1) initializes the system and calls the parser.

The parser (parsell1) directs the processing. It calls the scanner (lamscan1) to get tokens from the input. It calls the action routines (lamsem1) when it recognizes phrases in the input.

The scanner's file also contains procedures to insert input files under the direction of the *load* function.

The action routines (lamsem1) build the tree representation of the program and write out the results of expressions.

Copyright © 1996. Thomas W. Christopher

The interpreter (xeq1) handles the execution of a Lambda-1 program with the exception of the primitive functions, which are contained in the run-time system file (rts12). The recursive calls between xeq1 and rts12 suggests that they are part of the same module, but rts12 is kept in a separate file to allow it to be used in the Lambda-2 implementation as well.

The calls between modules are pictured in Figure 1.

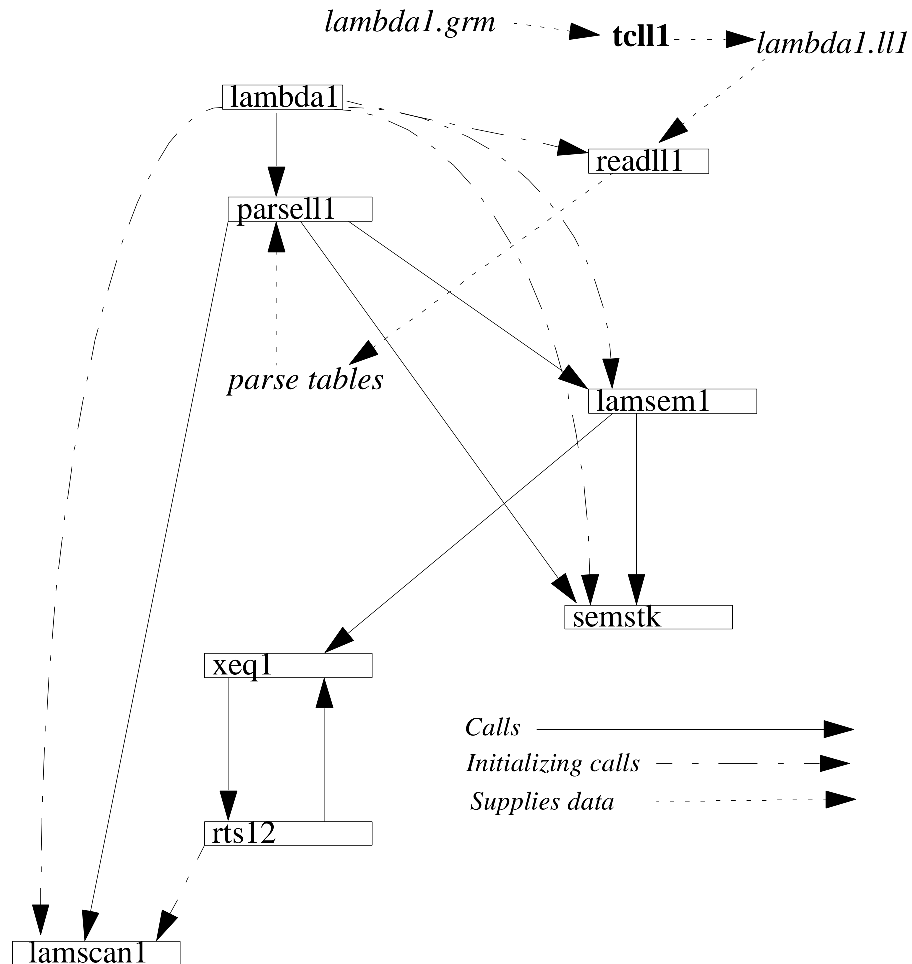


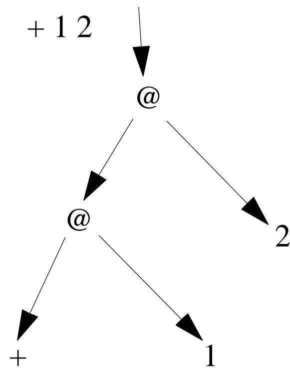
Figure 1 Module structure of Lambda 1.

1.3 Representation of Data Objects and Program Trees

The Lambda-1 interpreter uses the Icon data structures shown in Table 1. The code is represented by trees. Two such trees are shown in Figure 2, where App

 Copyright © 1996. Thomas W. Christopher

records are shown as @, Lambda records as λ, and Var records as the name of the variable.



def map=\f.\l.if (pair? l) (cons (f (hd l)) (map f (tl l))) l;

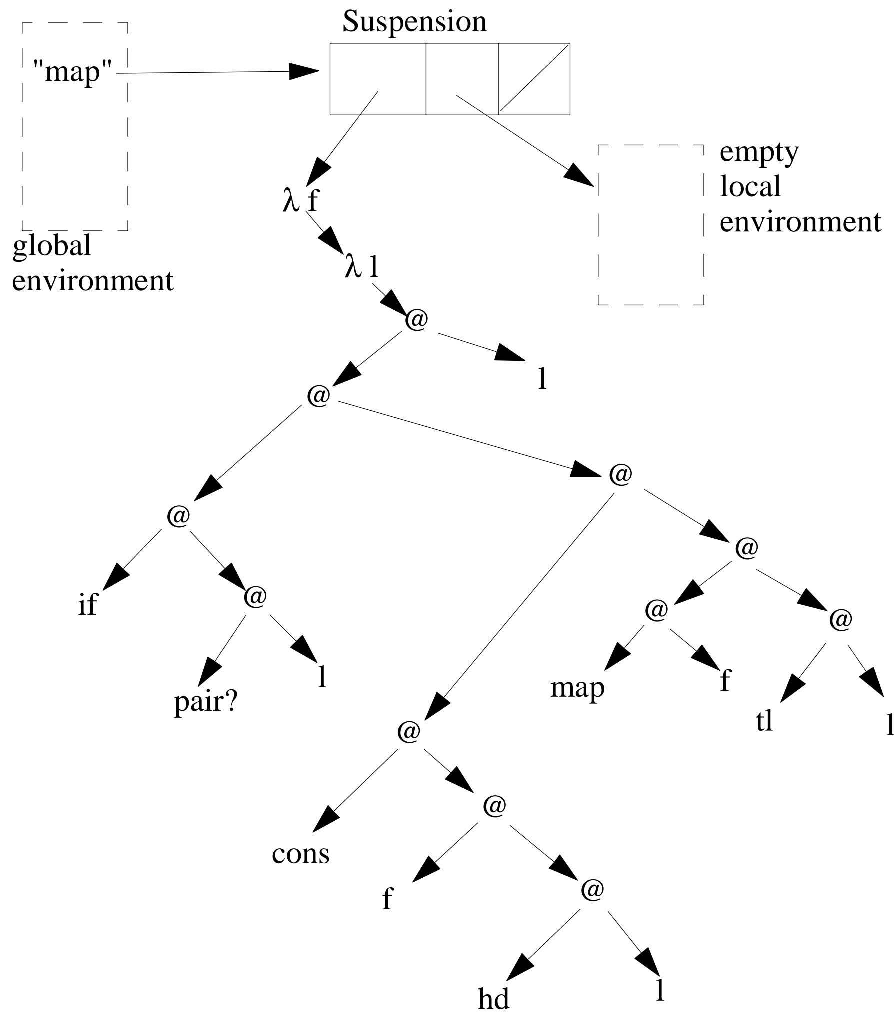


Figure 2 Example expression trees.

Copyright © 1996. Thomas W. Christopher

Table 1 Representation of Lambda-1.

Icon Data Type	Fields	Representing
`integer, real`		numeric
`string`		string
`table`		an environment, mapping variable names into values.
`record Boolean(name)`	`name` is `"true"` or `"false"`	represents a logical value. There are only two in existence, pointed to by global variables, `trueVal` and `falseVal`.
`record Error(msg)`	`msg` is a string describing the error	represents an error that the interpreter caught. (There are all too many errors that just crash Icon.)
`record Suspension( expr, env, value)`	`expr`—tree for an expression to evaluate. `env`—local environment in which to evaluate the expression. `value`—`&null` until the expression is evaluated, then it contains the value previously computed.	represents an unevaluated expression. All actual parameters are passed as suspensions. All definitions, `def name = expression` bind the `name` in the global environment to a suspension for the `expression`. When the value of the expression is needed, it is computed and stored in the `value` field from whence it is fetched when subsequently needed.
`record Lambda( var,expr)`	`var`—the bound variable. `expr`—the body of the function.	represents `\ var . expr`
`record Closure(fn,env)`	`fn`—a lambda expression. `env`—an environment in which to evaluate the function.	binds a lambda function `fn` to an environment so that if it is applied later it's expression will be evaluated in the surrounding local environment `env`.

 Copyright © 1996. Thomas W. Christopher

Table 1 Representation of Lambda-1.

Icon Data Type	Fields	Representing
`record Primitive( needed, fn)`	`needed`—the number of actual parameters the function needs. `fn`—the Icon function that implements the primitive function. These Icon functions are in file rts12.icn.	the built-in functions like `+`, `cons`, `if`,....
`record PrimitiveClosure( fn, args)`	`fn`—a Primitive record for the function. `args`—a list of the arguments the function has been applied to so far.	a built-in function that has been applied to fewer arguments than it requires. The arguments it has gathered so far are saved in the list args.
`record Var(name)`	`name`—a string, the name of the variable.	a variable. Variable names are not only identifiers, but also strings of special characters, like `"+"`.

`record App(fn,expr)`	`fn`—the subtree representing the function.
	
	`expr`—the subtree representing the argument.
		an application of a function to an argument. Since functions are Curried, function application associates to the left. E.g.
		
		`+ 1 2`
		
		is represented as
		
		`App(App(Var("+"),1),2)`

1.4 The Interpreter

Code for the interpreter is shown in Figure 3. The declarations up to line 34 are also expressed in Table 1.

The code from line 110 to the end deals with writing out the results of the expressions. Procedure *writeList* writes out the first several elements of a list and the sublists down a few levels. It doesn't try to write the entire list because Lambda-1 allows the creation of conceptionally infinite lists. It forces the evaluation of suspensions down to the maximum number of levels needed. Procedure *writeList* is only called for a *Pair* record.

Procedure *writeElem* is called to write out anything except a *Pair*.

Procedure *writeVal* is called to write out any sort of value. It dispatches to *writeElem* or *writeList*.

Copyright © 1996. Thomas W. Christopher

Procedure *force*, lines 96 to 107, is called to get a value from a subexpression. If the subexpression is anything other than a suspension, it is simply returned. If it is a suspension, there are two possibilities:

1. the value has already been computed and its value has been saved in field *value*. In this case, the value is simply returned.
2. the value has not yet been computed; field *value* is *&null*. In this case, field *expr* points to an expression to evaluate and field *env* points to a local environment. The expression is evaluated in the local environment, and the value is placed in the *value* field. Following this, the *expr* and *env* fields are no longer needed.

The real work of evaulation is done by procedure eval, lines 35 to 93. Parameters *expr* and *env* tell *eval* to evaluate expression *expr* in local environment *env*. Local variable *stk* is used to hold a pushdown of suspensions for the arguments to functions. The body of eval repeatedly examines the node type of *expr* and chooses what to do for each possibility. The repeat loop is allows *eval* to handle some forms of tail-end recursion without actually calling itself.

If the expression is an apply node, *App*, line 39, *eval* pushes a suspension for the argument on the stack and goes on to evaluate the left subtree in the current environment. Starting at the top of a function call, *eval* walks down the spine pushing suspensions on *stk*. When it finally gets to the function at the end of the spine, all the arguments, left to right, will be on the stack from top to bottom.

If the expression is a variable node, *Var*, line 43, eval tries to look up the value associated with the variable's name. It looks first in the local environment and then in the global environment.

If the node is a suspension, *Suspension*, line 48, *eval* calls *force* to get the value.

If the node is a *Lambda*, line 51, there are two possibilities, depending on whether there are any arguments on the stack.

1. If there are no arguments, eval returns a closure for the expression containing the lambda expression and the current environment.
2. If there is at least one argument, then the function can be called directly; *eval* creates a copy of the local environment and binds in it the name in the *var* field of the Lambda to the argument removed from the top of the stack. Then *eval* evaluates the expression part of the lambda in this new environment, by assinging the new environment to *env* and the expression to *expr* and looping.

If the expression to evaluate is a *Closure*, line 58, it contains a lambda expression and an environment. Again *eval*'s behavior depends on whether there is an argument for the function on the stack or whether the stack is empty.

1. If *stk* is empty, the closure is simply returned.
2. If there is an argument for the closure, eval makes a copy of the closure's environment and binds the lambda variable to the top argument from the

 Copyright © 1996. Thomas W. Christopher

stack. Then it evaluates the lambda's expression in the new environment.

If *eval* finds a primitive function, *Primitive*, line 75, it was the result of looking up a variable name. All the primitive functions are predefined in the global table. A Primitive has two fields, one telling how many arguments are needed, and the other pointing to an Icon procedure to preform the operation. If there are enough arguments for the function on the stack, the Icon procedure is called, passing it the stack and allowing it to remove its arguments itself. On the other hand, if there are not enough arguments available, *eval* returns a *PrimitiveClosure*, a closure for a primitive function which contains the function and all the arguments that were on the stack.

If *eval* finds a *PrimitiveClosure*, line 66, there are two possibilities:

1. If the number of arguments contained in the primitive closure plus the number of arguments currently on the stack are still too few for the primitive function, *eval* appends the stack to the end of the argument list in the primitive closure and returns it.
2. If there are enough arguments, the arguments from the primitive closure are placed on the stack and the stack is passed to the primitive function.

Finally, if *eval* finds an *Error* value, line 83, it simply returns it.

Figure 3 *Module* ***xeq1.icn****.*

```
 1 # Lambda calculus execution
 2 #  (SECD-like design)
 3 #
 4 record Lambda(var,expr)
 5      # var: name
 6      # function of *vars parameters
 7 record App(fn,expr)
 8      # apply fn to (a suspension of) expr
 9 record Var(name)
10      # a variable (name is a string)
11 record Suspension(expr,env,value)
12      # env: a table binding names to suspensions (or closures?)
13      # value: &null until evaluated, then the value
14      # evaluate expr in environment env
15 record Closure(fn,env)
16      # fn: the function to apply (lambda expression or primitive)
17      # env: a table binding names to suspensions (or closures?)
18 record PrimitiveClosure(fn,args)
19      # fn: the function to apply (lambda expression or primitive)
20      # args: list of arguments gathered so far
21 record Primitive(needed,fn)
22      # needed: int - number of arguments needed
23      # fn: procedure - Icon procedure for primitive function
24 record Error(msg)
25      # msg: string
```

Copyright © 1996. Thomas W. Christopher

```
26      # an error has been detected
27 record Boolean(name)
28 record Nil()
29 record Pair(hd,tl)
30
31 global trueVal,falseVal,nilVal
32
33 global globalEnv,nullEnv
34
35 procedure eval(expr,env)
36 local stk,i,s,c
37 stk:=[]
38 repeat case type(expr) of {
39   "App": {
40      push(stk,Suspension(expr.expr,env))
41      expr:=expr.fn
42   }
43   "Var": {
44      expr:= \env[expr.name] |
45              \globalEnv[expr.name] |
46              Error(expr.name||" is undefined")
47   }
48   "Suspension": {
49      expr:=force(expr)
50   }
51   "Lambda": {
52      if *stk=0 then
53              return Closure(expr,env)
54      env := copy(env)
55      env[ expr.var ] := pop(stk)
56      expr := expr.expr
57   }
58   "Closure": {
59      if *stk=0 then return expr
60      else {
61              env := copy(expr.env)
62              env[expr.fn.var] := pop(stk)
63              expr := expr.fn.expr
64      }
65   }
66   "PrimitiveClosure": {
67      if *expr.args + *stk < expr.fn.needed then {
68              expr:=PrimitiveClosure(expr.fn,expr.args ||| stk)
69              return expr
70      } else {
71              stk := expr.args ||| stk
72              expr := expr.fn.fn(stk)
73      }
74   }
```

 Copyright © 1996. Thomas W. Christopher

```
 75   "Primitive": {
 76     if expr.needed > *stk then {
 77             return PrimitiveClosure(expr,stk)
 78     } else {
 79             expr := expr.fn(stk)
 80             #....if *stk = 0 then return expr
 81     }
 82   }
 83   "Error": {
 84     return expr
 85   }
 86   default: {
 87     if *stk = 0 then return expr
 88     else return Error("eval: WHNF not value")
 89   }
 90
 91 }
 92
 93 end
 94
 95
 96 procedure force(arg)
 97 case type(arg) of {
 98    "Suspension": {
 99     if /arg.value then {
100             arg.value:=eval(arg.expr,arg.env)
101             arg.env := &null
102     }
103     return arg.value
104    }
105    default:return arg
106 }
107 end
108
109
110 procedure writeVal(v)
111 if type(v)=="Pair" then writeList(v)
112 else writeElem(v)
113 Write()
114 return
115 end
116
117 procedure writeElem(v)
118 case type(v) of {
119 "Boolean":Writes(v.name)
120 "Nil":     Writes("nil")
121 "Pair":    writeList(v)
122 "Error":Writes(v.msg)
123 "integer"|"real"|"string": Writes(v)
```

Copyright © 1996. Thomas W. Christopher

```
124 "Lambda"|"Closure"|
125     "Primitive"|"PrimitiveClosure":
126             Writes("a function")
127 "Suspension": Writes("an unevaluated expression")
128 default: Writes("unknown value")
129 }
130 return
131 end
132
133
134 procedure writeList(v,d)
135 local h,i
136 /d := 2
137 h:=force(v.hd)
138 if type(h)=="Pair" then {
139     if d>0 then {
140             Writes("(")
141             writeList(h,d-1)
142             Writes(")")
143     } else Writes("(...)")
144 } else {
145     writeElem(h)
146 }
147 i := 5
148 while i>0 do {
149     Writes(":")
150     v:=force(v.tl)
151     if type(v)~=="Pair" then return writeElem(v)
152     else {
153             h:=force(v.hd)
154             if type(h)=="Pair" then {
155               if d>0 then {
156                     Writes("("); writeList(h,d-1); Writes(")")
157               } else Writes("(...)")
158             } else {
159                     writeElem(h)
160             }
161     }
162     i-:=1
163 }
164 if type(v)=="Pair" then Writes(":....")
165 else {Writes(":");writeElem(v)}
166 end
167
168 procedure Writes(L[])
169 every writes(!L)
170 \logFile & every writes(logFile,!L)
171 return
172 end
```

 Copyright © 1996. Thomas W. Christopher

```
173
174 procedure Write(L[])
175 Writes!put(L,"\n")
176 return
177 end
```

Copyright © 1996. Thomas W. Christopher

Chapter 2 Tour of Lambda 2

2.1 Files

The files related to Lambda-2 are

- build2.bat—a batch file to build the Lambda-2 compiler/interpreter.
- lambda2.grm—the grammar (in TCLL1 form) for the Lambda-2 parser.
- lambda2.ll1—the parse tables resulting from passing lambda2.grm through TCLL1.
- lambda2.icn—the main program for Lambda-2.
- lamscan2.icn—the scanner for Lambda-2.
- lamsem2.icn—the semantics (action) routines for Lambda-2. These build a tree representation for the Lambda-2 definitions and expressions.
- xeq2.icn—the routines to interpret the tree representations expressions. This file contains the definitions of the data structures used for both the program tree and data structures.
- rts12.icn—the routines to execute the primitive functions used by both Lambda-1 and Lambda-2.
- parsell1.icn, readll1.icn, and semstk.icn—routines from the TCLL1 parser.

 Copyright © 1996. Thomas W. Christopher

2.2 Program Structure

The procedure calls between modules in Lambda-2 is shown in Figure 4.

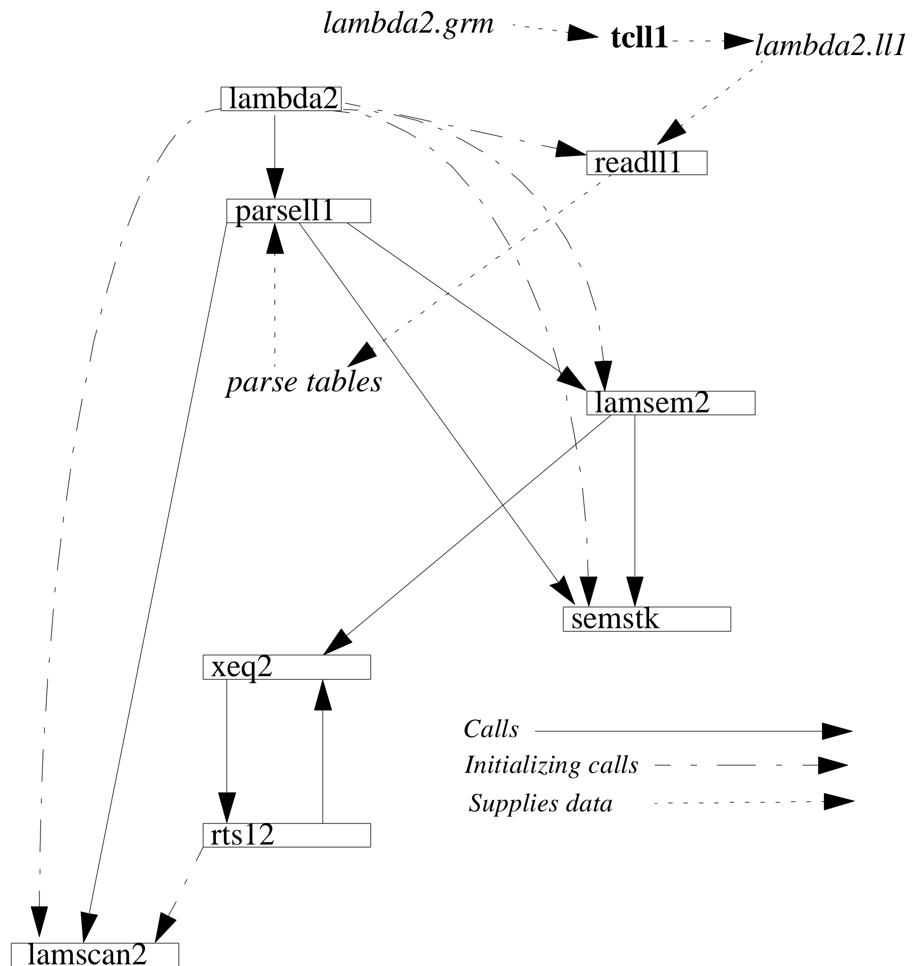


Figure 4 Module structure of Lambda-2.

2.3 Representation of Data Objects and Program Trees

The Icon data types used to implement Lambda-2 data types and program trees are listed in Table 2.

Table 2 Representation of Lambda-2.

Icon Data Type	Fields	Representing
`integer, real`		numeric
`string`		string

Copyright © 1996. Thomas W. Christopher

Table 2 Representation of Lambda-2.

Icon Data Type	Fields	Representing
`table`		an environment, mapping variable names into values.
`record Boolean(name)`	`name` is `"true"` or `"false"`	represents a logical value. There are only two in existence, pointed to by global variables, `trueVal` and `falseVal`.
`record Error(msg)`	`msg` is a string describing the error	represents an error that the interpreter caught. (There are all too many errors that just crash Icon.)
`record Suspension( expr, env, value)`	`expr`—tree for an expression to evaluate. `env`—local environment in which to evaluate the expression. `value`—`&null` until the expression is evaluated, then it contains the value previously computed.	represents an unevaluated expression. All actual parameters are passed as suspensions. All definitions, `def name = expression` bind the `name` in the global environment to a suspension for the `expression`. When the value of the expression is needed, it is computed and stored in the `value` field from whence it is fetched when subsequently needed.
`record Lambda( vars,expr)`	`vars`—a list of the bound variables. `expr`—the body of the function.	represents `\` v_1 v_2 ... v_n `. expr`
`record Closure( needed, fn, args, env)`	`needed`—the number of actual parameters the function still needs. `fn`—a lambda expression. `args`—a list of the arguments the function has been applied to so far. `env`—an environment in which to evaluate the function.	binds a function `fn` (either a lambda or a primitive) to an environment so that if it is applied later its expression will be evaluated in the surrounding local environment `env`. The arguments to which the function has already been applied are in the list `args`. The number of arguments still needed is in field `needed`.

 Copyright © 1996. Thomas W. Christopher

Table 2 Representation of Lambda-2.

Icon Data Type	Fields	Representing
`record Primitive( needed, fn)`	`needed`—the number of actual parameters the function needs. `fn`—the Icon function that implements the primitive function. These Icon functions are in file rts12.icn.	the built-in functions like `+`, `cons`, `if`,....
`record Var(name)`	`name`—a string, the name of the variable.	a variable. Variable names are not only identifiers, but also strings of special characters, like `"+"`.

`record App(fn,expr)`	`fn`—the subtree representing the function.
	
	`expr`—the subtree representing the argument.
		an application of a function to an argument. Since functions are Curried, function application associates to the left. E.g.
		
		`+ 1 2`
		
		is represented as
		
		`App(App(Var("+"),1),2)`

`record If(expr, truePart, falsePart)`	for the if-expression: if expr then truePart else falsePart `expr`—the control expression, always evaluated `truePart`—the expression to be evaluated if `expr` is true `falsePart`—the expression to be evaluated if `expr` is false	this expression node replaces the `if` function of Lambda-1. The replacement is solely for reasons of efficiency. The `truePart` and `falsePart` expressions would be passed as suspensions to the `if` function, which would require a recursive call of the *eval* procedure in the interpreter. The If node, however, is handled directly. If the expression evaluates true, the If node is replaced by the `truePart`; if false, it is replaced by the `falsePart`.

Copyright © 1996. Thomas W. Christopher

Table 2 Representation of Lambda-2.

Icon Data Type	Fields	Representing
`record Let( vars, exprs, expr)`	for the let-expression let n_1=e_1; n_2=e_2; ... n_m=e_m in e `vars`—an Icon list of the bound names: [n_1,n_2,...,n_m]. `exprs`—an Icon list of the expressions bound to the names: [e_1,e_2,...,e_m]. `expr`—the body of the let-expression, e.	a *let* expression: evaluate `expr` in the environment `E'` formed from the current local environment `E` by binding each name in `vars` to a suspension for the corresponding expression in `exprs` bound in environment `E`. This is really unnecessary. let n_1=e_1; n_2=e_2; ... n_m=e_m in e is equivalent to (\n_1 n_2 n_m .e)e_1 e_2 e_m
`record LetRec( vars, exprs, expr)`	for the letrec-expression letrec n_1=e_1; n_2=e_2; ... n_m=e_m in e `vars`—an Icon list of the bound names: [n_1,n_2,...,n_m]. `exprs`—an Icon list of the expressions bound to the names: [e_1,e_2,...,e_m]. `expr`—the body of the let-expression, e.	a *recursive let* expression: evaluate `expr` in the environment `E'` formed from the current local environment `E` by binding each name in `vars` to a suspension for the corresponding expression in `exprs` bound in environment `E'`; that is, all the new names are bound to the environment in which they are defined and hence can see each other.

 Copyright © 1996. Thomas W. Christopher

A Lambda-2 code tree for the function is pictured in Figure 5.

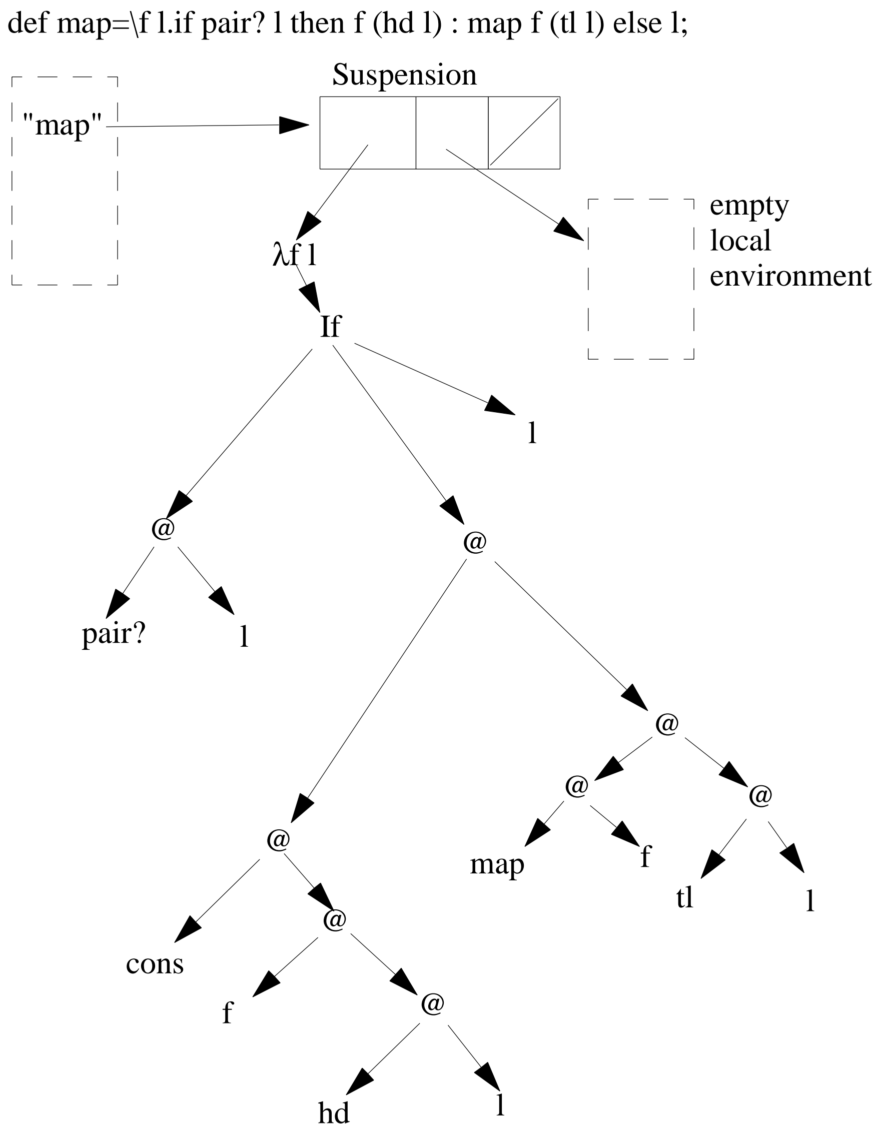


Figure 5 Example code tree in Lambda-2.

2.4 The Interpreter

Code for the interpreter is shown in Figure 6. The declarations up to line 39 are also expressed in Table 2.

The code from line 158 to the end deals with writing out the results of the expressions. Procedure *writeList* writes out the first several elements of a list and the sublists down a few levels. It doesn't try to write the entire list because Lambda-2 allows the creation of conceptually infinite lists. It forces the evaluation of suspensions down to the maximum number of levels needed. Procedure *writeList* is only called for a *Pair* record.

Copyright © 1996. Thomas W. Christopher

Procedure *writeElem* is called to write out anything except a *Pair*.

Procedure *writeVal* is called to write out any sort of value. It dispatches to *writeElem* or *writeList*.

Procedure *force*, lines 144 to 155, is called to get a value from a subexpression. If the subexpression is anything other than a suspension, it is simply returned. If it is a suspension, there are two possibilities:

1. the value has already been computed and its value has been saved in field *value*. In this case, the value is simply returned.
2. the value has not yet been computed; field *value* is *&null*. In this case, field *expr* points to an expression to evaluate and field *env* points to a local environment. The expression is evaluated in the local environment, and the value is placed in the *value* field. Following this, the *expr* and *env* fields are no longer needed.

Procedure *bind*, lines 132 to 142, differs from Lambda-1. The procedure represents an optimization. Where Lambda-1 would simply create a suspension, Lambda-2 calls bind which checks to see what kind of expression is being suspended. Procedure *bind* takes two parameters, *expr* and *env*. There are four cases:

- if *expr* is a variable, bind looks up its value in the environments *env* and the global environment.
- if *expr* is a lambda expression, *bind* creates a closure for it.
- if *expr* is a constant (integer, real, string, or primitive function), *bind* just returns it as is.
- otherwise, *bind* returns a suspension.

Procedure *eval* calls *bind* both to push the parameters of a function on the stack and to bind the values to names in a let-expression.

To bind the values to names in a letrec-expression, *eval* calls *bindrec(expr,env)*. Procedure *bindrec* is like bind in that it creates a closure for a lambda expression and returns constants as is. Unlike bind, it does not try looking up the values of variables: those in the local environment may not have been inserted into the table yet.

The real work of evaluation is done by procedure *eval*, lines 46 to 121. Parameters *expr* and *env* tell *eval* to evaluate expression *expr* in local environment *env*. Local variable *stk* is used to hold a pushdown of suspensions for the arguments to functions. The body of *eval* repeatedly examines the node type of *expr* and chooses what to do for each possibility. The repeat loop is allows *eval* to handle some forms of tail-end recursion without actually calling itself.

If the expression is an apply node, *App*, line 50, *eval* binds the argument in the current environment, pushes it on the stack and goes on to evaluate the left subtree in the current environment. Starting at the top of a function call, *eval* walks

 Copyright © 1996. Thomas W. Christopher

down the spine pushing bindings of the arguments on *stk*. When it finally gets to the function at the end of the spine, all the arguments, left to right, will be on the stack from top to bottom.

If the expression is a variable node, *Var*, line 70, *eval* tries to look up the value associated with the variable's name. It looks first in the local environment and then in the global environment.

If the node is a suspension, *Suspension*, line 81, *eval* calls *force* to get the value.

If the node is a *Lambda*, line 84, there are two possibilities, depending on whether there are any arguments on the stack.

1. If there are too few arguments, *eval* returns a closure for the expression containing the lambda expression, the arguments already supplied, and the current environment. The closure indicates the number of parameters still needed.
2. If there are at least as many arguments as the function requires, the function can be called directly; *eval* creates a copy of the local environment and inserts in it the names in the *vars* field of the Lambda bound to the arguments removed from the top of the stack. Then *eval* evaluates the expression part of the lambda in this new environment, by assigning the new environment to *env* and the expression to *expr* and looping.

If the expression to evaluate is a *Closure*, line 91, it contains a lambda expression and an environment. Again *eval*'s behavior depends on whether there are enough arguments for the function on the stack or not.

1. If *stk* is contains too few arguments to call the function, the closure is copied, any additional arguments are removed from the stack and added to the copy, and the copy is returned.
2. If there are as many arguments as the closure needs, *eval* makes a copy of the closure's environment and binds the lambda variables to the top arguments from the closure and the stack. Then it evaluates the lambda's expression in the new environment.

If *eval* finds a primitive function, *Primitive*, line 103, it has the same two options as for a lambda expression: if there are enough arguments, it evaluates the function. If there are too few, it creates a closure which contains the function and all the arguments that were on the stack. Unlike Lambda-1, there is no *Primitive-Closure* object: both lambda expressions and primitive functions use the same closures.

If *eval* finds a let-expression, *Let*, line 60, it creates a new environment, initially a copy of the current environment. *Eval* inserts into the new environment every let-variable associated with its corresponding expression. The expressions, however, are bound to the current environment so they can't see themselves or any of the other variables. Then *eval* loops to evaluate the body expression in the new environment.

Copyright © 1996. Thomas W. Christopher

If *eval* finds a letrec-expression, *LetRec*, line 54, it creates a new environment, initially a copy of the current environment. *Eval* inserts into the new environment every letrec-variable associated with its corresponding expression. The expressions are bound in the new environment so they can see themselves and all of the other variables. Then *eval* loops to evaluate the body expression in the new environment.

If *eval* finds an if-expression, *If*, line 75, it calls itself recursively to evaluate the control expression. If the control expression returns true, *eval* loops to evaluate the true part; if false, the false part; and if anything else, it returns an error.

Finally, if *eval* finds an *Error* value, line 111, it simply returns it.

Figure 6 *Module* ***xeq2.icn***.

```
 1 # Lambda calculus execution
 2 #  (Home made design)
 3 #
 4 record Lambda(vars,expr)
 5      # vars: [name, ...]
 6      # function of *vars parameters
 7 record Let(vars,exprs,expr)
 8      # vars: [name, ...]
 9      # exprs: [expr, ...]
10      # successively bind vars[i] to a suspension of exprs[i],
11      #   i:=1 to *vars
12 record LetRec(vars,exprs,expr)
13      # vars: [name, ...]
14      # exprs: [expr, ...]
15      # simultaneously bind vars[i] to a suspension of exprs[i],
16 record App(fn,expr)
17      # apply fn to (a suspension of) expr
18 record If(expr,truePart,falsePart)
19      # if the expr evaluates true, evaluate truePart else falsePart
20 record Var(name)
21      # a variable (name is a string)
22 record Suspension(expr,env,value)
23      # env: a table binding names to suspensions (or closures?)
24      # value: &null until evaluated, then the value
25      # evaluate expr in environment env
26 record Closure(needed,fn,args,env)
27      # needed: int - the number of parameters needed
28      # fn: the function to apply (lambda expression or primitive)
29      # args: [arg, ...] - the already accumulated arguments
30      # env: a table binding names to suspensions (or closures?)
31 record Primitive(needed,fn)
32      # needed: int - number of arguments needed
33      # fn: procedure - Icon procedure for primitive function
34 record Error(msg)
35      # msg: string
36      # an error has been detected
37 record Boolean(name)
38 record Nil()
39 record Pair(hd,tl)
40
41 global trueVal,falseVal,nilVal
42
43 global globalEnv,nullEnv
44
```

 Copyright © 1996. Thomas W. Christopher

```
 45
 46 procedure eval(expr,env)
 47 local stk,i,s,c,newenv,t
 48 stk:=[]
 49 repeat case type(expr) of {
 50   "App": {
 51      push(stk,bind(expr.expr,env))
 52      expr:=expr.fn
 53   }
 54   "LetRec": {
 55      env:=copy(env)
 56      every i:=1 to *expr.vars do
 57         env[expr.vars[i]]:=bindrec(expr.exprs[i],env)
 58      expr:=expr.expr
 59   }
 60   "Let": {
 61      newenv:=copy(env)
 62      every i:=1 to *expr.vars do {
 63         s:=bind(expr.exprs[i],env)
 64            # for let*: env:=copy(env)
 65         newenv[expr.vars[i]]:=s
 66      }
 67      env:=newenv
 68      expr:=expr.expr
 69   }
 70   "Var": {
 71      expr:= \env[expr.name] |
 72         \globalEnv[expr.name] |
 73         Error(expr.name||" is undefined")
 74   }
 75   "If": {
 76      t:=eval(expr.expr,env)
 77      if t===trueVal then expr:=expr.truePart
 78      else if t===falseVal then expr:=expr.falsePart
 79      else  return Error("if: not a Boolean")
 80   }
 81   "Suspension": {
 82      expr:=force(expr)
 83   }
 84   "Lambda": {
 85      if *expr.vars > *stk then
 86         return Closure(*expr.vars - *stk,expr,stk,env)
 87      env := copy(env)
 88      every i:=!expr.vars do env[i] := pop(stk)
 89      expr := expr.expr
 90   }
 91   "Closure": {
 92      if expr.needed > *stk then {
 93         c := copy(expr)
 94         c.needed -:= *stk
 95         c.args := expr.args ||| stk
 96         return c
 97      } else {
 98         stk := expr.args|||stk
 99         env := expr.env
100         expr := expr.fn
101      }
102   }
103   "Primitive": {
104      if expr.needed > *stk then {
105         return Closure(expr.needed - *stk,expr,stk,env)
106      } else {
107         expr := expr.fn(stk)
108         #....if *stk = 0 then return expr
```

Copyright © 1996. Thomas W. Christopher

```
109      }
110   }
111   "Error": {
112      return expr
113   }
114   default: {
115      if *stk = 0 then return expr
116      else return Error("eval: WHNF not value")
117   }
118
119 }
120
121 end
122
123 procedure bindrec(expr,env)
124 case type(expr) of {
125 "Lambda": return Closure(*expr.vars,expr,[],env)
126 "integer"|"real"|"string"|"Primitive":
127         return expr
128 default:return Suspension(expr,env)
129 }
130 end
131
132 procedure bind(expr,env)
133 case type(expr) of {
134 "Var":   return \env[expr.name] |
135             \globalEnv[expr.name] |
136             Error(expr.name||" is undefined")
137 "Lambda": return Closure(*expr.vars,expr,[],env)
138 "integer"|"real"|"string"|"Primitive":
139         return expr
140 default:return Suspension(expr,env)
141 }
142 end
143
144 procedure force(arg)
145 case type(arg) of {
146    "Suspension": {
147      if /arg.value then {
148         arg.value:=eval(arg.expr,arg.env)
149         arg.expr := arg.env := &null
150      }
151      return arg.value
152    }
153    default:return arg
154 }
155 end
156
157
158 procedure writeVal(v)
159 if type(v)=="Pair" then writeList(v)
160 else writeElem(v)
161 Write()
162 return
163 end
164
165 procedure writeElem(v)
166 case type(v) of {
167 "Boolean":Writes(v.name)
168 "Nil":   Writes("nil")
169 "Pair":  writeList(v)
170 "Error":Writes(v.msg)
171 "integer"|"real"|"string": Writes(v)
172 "Lambda"|"Closure"|
```

 Copyright © 1996. Thomas W. Christopher

```
173      "Primitive"|"PrimitiveClosure":
174          Writes("a function")
175 "Suspension": Writes("an unevaluated expression")
176 default: Writes("unknown value")
177 }
178 return
179 end
180
181
182 procedure writeList(v,d)
183 local h,i
184 /d := 2
185 h:=force(v.hd)
186 if type(h)=="Pair" then {
187      if d>0 then {
188          Writes("(")
189          writeList(h,d-1)
190          Writes(")")
191      } else Writes("(...)")
192 } else {
193      writeElem(h)
194 }
195 i := 10
196 while i>0 do {
197      Writes(":")
198      v:=force(v.tl)
199      if type(v)~=="Pair" then return writeElem(v)
200      else {
201          h:=force(v.hd)
202          if type(h)=="Pair" then {
203             if d>0 then {
204             Writes("("); writeList(h,d-1); Writes(")")
205             } else Writes("(...)")
206          } else {
207             writeElem(h)
208          }
209      }
210      i-:=1
211 }
212 if type(v)=="Pair" then Writes(":....")
213 else {Writes(":");writeElem(v)}
214 end
215
216 procedure Writes(L[])
217 every writes(!L)
218 \logFile & every writes(logFile,!L)
219 return
220 end
221
222 procedure Write(L[])
223 Writes!put(L,"\n")
224 return
225 end
```

Copyright © 1996. Thomas W. Christopher

Chapter 3 Implementation projects

3.1 Infix binary operators

Modify Lambda-2 to put in binary operators.

The Lambda-2 functions

+ - * / mod < <= = ~= > >= hd tl

now become binary operators. For example

+ a (- (* b c) d) becomes a+b*c-d

An operator may be converted into a function by enclosing it in parentheses, for example:

def inc = (+) 1;

The minus sign is both binary and unary; enclosing it in parentheses produces the function for the binary minus. We can define the unary negation function as:

def neg = (-) 0;

Function applications have higher precedence than all the operators except hd and tl. For example, map can be defined as:

def map = \f l.if null? l then nil else f hd l: map f tl l;

and map2 can be defined as

def map2 = \f l m.if null? l or null? m then nil else f hd l hd m: map2 f tl l tl m;

The precedence and associativity of the operators are given in Table 3.

Table 3 Table of operators.

Precedence	Operators	Associates	Notes
highest	`hd tl`	unary	
	function application		
	`-`	unary	- a is equivalent to (-) 0 a

 Copyright © 1996. Thomas W. Christopher

Table 3 Table of operators.

Precedence	Operators	Associates	Notes
	`* / mod` `+ -`	left left	
	`< <= = ~= >= >`	non-associative	
	`not`	unary	not a is equivalent to if a then false else true
	`and` `or`	right right	a and b is equivalent to if a then b else false a or b is equivalent to if a then true else b
lowest	`:`	right	already implemented

Suggested approach

1) Modify the Lambda-2 grammar to include these operators and the parenthesized forms. Debug the grammar by passing it through TCLL1. Include action symbols.

2) Modify the Lambda-2 scanner to recognize these operators.

3) Write action routines to translate the new expressions. The best translation is to the current Lambda-2 internal form, rather than inventing new internal forms, e.g. (a+b) becomes the same as the current (+ a b) (or the new ((+) a b)) . That way you don't have to change the interpreter.

Copyright © 1996. Thomas W. Christopher