First, A Few Words About Bindings, Scope, Environment

• A binding is the association of a value to an identifier (aka “variable”).

• The purpose of binding is to be able to use the identifier in place of the value.
  • The value may be messy to write/read.
  • The value may be determined by some computation.

high-level functional programming

bob keller

september 2010

let in racket

• One means of establishing a binding is to use a ‘let expression:

\[
\text{let (}
\begin{align*}
\text{freq} &= \frac{1}{1 \times 2 \pi} \\
\text{(* 3 freq)}
\end{align*}
\text{)}
\]

think of this as an equation establishing a binding:

- \(\text{LHS} = \text{RHS}\)
- \(\text{freq} = \frac{1}{1 \times 2 \pi}\)

This expression is evaluated to get the result of the let. It may use the binding.

(let (  
  (freq (/ 1 (* 2 pi)))  
)  
(* 3 freq))

special forms

• A special form in Racket is not a function being applied.

• Certain identifiers are have a special built-in meaning, that signals a special form.

• Examples of such identifiers are:
  - define, let, let*, if, cond, case, lambda

the ‘let special form

(let (  
  ... 0 or more equations ...
  )  
... result expression ...
)

The reason for the red parens is that there can be more than one equation. These parens group the equations, separating them from the result expression.

environments

• An environment is a set of bindings.
• The let expression effectively establishes an environment for evaluation of the result expression.

(let (  
  ... 0 or more equations ...
  )  
... result expression ...
)

result expression can use bindings in the inner environment

bindings determined in equations augment those in the outer environment of the let expression
Outer vs. Inner Environments

Suppose that a is bound to 5 outside.

(let
  (b 3)
  (c (* 2 a))
  (+ a b c)
)  

Result is:

A Tricky Thing

• The right-hand sides of the equations
  in a 'let expression are evaluated in
  the outer environment.

• Any shadowing of a binding
  (re-binding an identifier) are not seen
  in that environment.

Tricky Environments Illustrated

Suppose that a is bound to 5 outside.

(let
  (a 3)
  (c (* 2 a))
  (+ a c)
)  

Result is:

Scope

• By the scope of an identifier, we mean the
  set of places where that identifier has its
  given meaning.

(let
  (a 3)
  (c (* 2 a))
  (+ a c)
)  

The scope of this 'a is
the result expression only, not the RHS of the next equation.

let* vs. let

• let* is another special form. It has the same
  syntax as let, except for let* vs. let.

• In let* the environment is augmented with
  each new equation.

• The scope of a variable includes the RHS of
  equations that follow and the result
  expression (unless shadowing occurs).

let* example

(let* (a 3)
  (c (* 2 a))
  (+ a c))

The scope of this 'a is
the result expression and the
RHS of the next equation.

The environment here is not the outer environment, but rather the outer environment
with a rebound to 3.

Result is:
Shadowing is Non-Destructive

- Although a variable may be shadowed, by re-binding in an inner environment, it retains its original value in the outer environment.
- Returning to that environment finds the variable to have the same binding.
- This is called "referential transparency".
- This is why definitions, such as those provided by let, are not regarded as assignment statements.

let* as nested lets

- (let* ( (v1 e1) (v2 e2) ... (vn en) ) result)
  is same as
- (let ( (v1 e1) )
  (let ( (v2 e2) )
  ...
  (let ( (vn en) ) result) ...) )

Why do I start with high-level programming rather than recursion?

- Recursion is not needed to understand the effect of a function.
- High-level programming is often a simpler way to achieve a given goal.
- We’ll get to recursion soon enough.

Two Ways to Define Factorial

- (define (fac n)
  (if (< n 2) 1
  (* n (fac (- n 1)))))
  [Assuming range is already defined.]
- (define (fac n)
  (foldl * 1 (range 1 n)))
  Can understand this definition in two pieces: foldl, range.

Why Functional Programming is So Important

- No side effects:
  - Easier to debug
  - Easier to get parallel execution (e.g. on a “multi-core” system)
- Composability:
  Easier to compose complex functions from simpler ones

Composability
Composability

More Ways to Compose

to get still more functions

Engineering Applications

“But can I get a job?”

from various recruiting websites
(Increasingly) Popular Functional Languages

- Haskell
- Erlang
- OCaml
- Scala
- F#
- Clojure

So should I become a functional programmer?

- Yes, but *don't stop there*.
- Have functional skills in your toolkit, but be able to work outside that domain as well.
- Be "amphibious" and "agnostic" and exploit the best of what the community has to offer.

mapping over a list

- map applies a 1-ary function to each element of a list, returning a list of the same length, with the results of the applications in order

> (define (cube x) (* x x x))
> (map cube '(1 2 3 4 5))
(1 8 27 64 125)

More mapping examples

> (map symbol->string '(I should care))
("I" "should" "care")

> (map string-length (map symbol->string '(I should care)))
(1 6 4)

> (map reverse '( (Washington George) (Lincoln Abraham) (Jefferson Thomas) (Obama Barack)))
((George Washington) (Abraham Lincoln) (Thomas Jefferson) (Barack Obama))

The Type of map

- $(\text{map } \text{Function} \times \text{List}) \rightarrow \text{List}$

- So map: $(\text{A} \rightarrow \text{B}) \times \text{A}^* \rightarrow \text{B}^*$

- A and B could be the same type

- map preserves the length of its argument list

n-ary map

- map will apply an n-ary function to n equal-length lists "pointwise"

> (map + '(1 2 3 4) '(9 8 7 6))
(10 10 10 10)

> (map list '(1 2 3 4) '(9 8 7 6))
((1 9) (2 8) (3 7) (4 6))
The type of n-ary map

- map: \( (A^n \to B) \times (A^*)^n \to B^* \)
- All argument lists must be the same length in Racket

foldr and foldl

- These functions "fold" a list into something like an element of the list.
- The first argument is a 2-ary function.
- The second argument is the result for an empty list.
- The third argument is the list being folded.

\[
\begin{align*}
\text{(define (demo x y) (list '+ x y))} \\
\text{> (foldl demo 0 '(1 2 3 4 5))} \\
\text{> (foldr demo 0 '(1 2 3 4 5))} \\
\text{> (define (demo x y) (list '+ x y))} \\
\text{> (foldl demo 0 '(1 2 3 4 5))} \\
\text{> (foldr demo 0 '(1 2 3 4 5))} \\
\end{align*}
\]

The type of foldl/foldr

- foldl: Function Unit List) Element
  \( A \times A \to A \\
  A \times A^* \to A \)
- So foldl: \((A \times A \to A) \times A \times A^* \to A\)
  meaning the set of 2-ary functions on A

Composing map and foldl

Function being mapped

fold vs. reduce

- reduce is the functional programming identifier for folding when the folding direction is non-specific.
- reduce is not in Racket with that name.
- map/reduce in combination achieved fame from Google’s many uses of it.
- Google recently patented its use of map/reduce!?
Reactions

Open Ended
# Open source and programming

Google’s MapReduce patent: what does it mean for Hadoop?

Hadoop

From Wikipedia, the free encyclopedia

Apache Hadoop is a Java software framework that supports data-intensive distributed applications under a free license. It enables applications to work with thousands of nodes and petabytes of data. Hadoop was inspired by Google’s MapReduce and Google File System (GFS) papers.

Hadoop is a top-level Apache project, being built and used by a community of contributors from all over the world. Hadoop has been the largest contributor to the project and uses Hadoop extensively in its web search and advertising businesses. IBM and Google have announced a major initiative to use Hadoop to support university courses in distributed computer programming.

Hadoop was created by Doug Cutting (now a Cloudera employee), who named it after his son’s stuffed elephant. It was originally developed to support distributed search for the Hotbot search engine project.

Our own map–reduce

> (define (map-reduce binary unit unary L)
  (foldr binary unit (map unary L))
)

> (map-reduce + 0 length '((1 2 3) (4 5) (6) ()))
6

Averaging a Non–Empty List

> (define (average L) (/ (foldl + 0 L) (length L)))

> (average '(1 2 3 4 5 6 7 8 9))
5

> (map average '((1 2 3) (2 3 4) (3 4 5) (5 6 7)))
(2 3 4 6)

> (average (map length '((1 2 3) (2 3 4) (3 4 5) (5 6 7))))
3

Filtering a List

• Function filter keeps elements that satisfy a predicate argument.

> (define (even x) (= 0 (modulo x 2))) ;; = is a numeric test

> (filter even '(1 2 3 4 5 7 9 10))
(2 4 10)

Example: Generalized Anagram

• Suppose spaces “don’t count”.

• How would you define function anagram?

• Filter out the spaces.
Anonymous Functions

• We functions don’t need names to do our job.

• You can define us anonymously to:
  • Avoid having to think up names
  • Avoid cluttering up the namespace with temporary functions

An Anonymous Function Doing the Same Job

A Named Function

(3 7 4 9 …) square-each (9 49 16 81 …)

Functions as “First-Class Citizens”

Disposition of data types in a typical language

<table>
<thead>
<tr>
<th>Type</th>
<th>Need name?</th>
<th>Use as argument</th>
<th>Return as result</th>
<th>Put in structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>String</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Function</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

In Racket, the answer to these is Yes

Lambda Expressions

• One way to specify an anonymous function uses the idea of “lambda expression”

  (lambda (x) …)

  means

  “the function that, with argument x, returns the value computed by …”

Lambda Expression Examples

• (lambda (x) (+ 5 x))
  “the function that adds 5 to its argument”

• (lambda (x y) (expt y x))
  “the function that raises its second argument y to the power of the first argument x”

• (lambda (x) (list x x))
  “the function that makes a 2–element list of its argument twice in succession”
Lambda Expressions are applied just like any other function

> (lambda (x) (+ 5 x)) 99
104
> (lambda (x y) (expt y x)) 10 2
1024
> (lambda (x) (list x x)) "foobar"
("foobar" "foobar")

mapping and filtering with lambda expressions

• This kind of usage is very common:

> (map (lambda (x) (* x x)) '(1 2 3 4 5))
(1 4 9 16 25)
> (filter (lambda (x) (> x 3)) '(1 2 3 4 5 6))
(4 5 6)

Imported Variables in Lambda Expressions

• By "imported" I mean variables that are not arguments. These are sometimes called “free variables” in the lambda expression.

• These variables retain the meaning they had at the time of the function’s definition. This is called static scope.

• They do not change their meaning based on context (which would be dynamic scope).

Example of Imported Variable

• Below, b is imported into the lambda expression.

> (let ((b 99))
  (map (lambda (x) (+ x b)) '(1 2 3 4 5)) )
(100 101 102 103 104)

Imported Values Bind Statically in Racket

> (let* ((b 99)
         (f (lambda (x) (+ b x)))
         (lambda (b) (f 1)) 1000)
100

Functions should not be chameleons.

Early implementations of Lisp tended to get this wrong. It was called the “Funarg problem”. [Google it]
**How Static Binding is Implemented**

- The compiler turns a function into a "closure".
- Racket shows closures cryptically, as `<procedure>`.
- A closure contains:
  - Reference to imported values
  - Code for evaluating the function, given the arguments
- Closures live "on the heap". They do not disappear when the stack shrinks.

**Racket tries to do “the right thing” for the user’s convenience**

```racket
> (lambda (x) (+ 5 x))
#<procedure>
```

```racket
> (define f (lambda (x) ( + x y)))
> (define y 5)
> (f 10)
15
```

```racket
> (let ( (y 100) )
> (f 10))
15
```

**Functions Returning Functions**

- (add n) returns a function, for any argument n

```racket
> (define (add n) (lambda (x) (+ x n)))
> (map (add 5) '(1 2 3 4 5))
(6 7 8 9 10)
> (map (add 10) '(1 2 3 4 5))
(11 12 13 14 15)
```

**The Same Definition Not Using Lambda**

This is called “Currying” the add function [Google this],
in honor of logician Haskell B. Curry.
The idea is due to Moses Schönfinkel .
Linguists called it “Schönfinkelsisation”.

**The Type of (add 5) in Racket**

```racket
> (add 5)
#<procedure>
```

**Functions that both take and give functions as arguments**

Returns a function that applies its argument twice in succession. (Not related to numeric doubling.)

```racket
> (define (double f) (lambda (x) (f (f x))))
> (define (square x) (* x x))
> ((double square) 5)
625
```
In Pictures

(double double)

Is this meaningful?

What would it do?

(double double)

How Big?

(((double (double double)) square) 5)

How Big?

Composing Functions Using Functions

> (double double)
#<procedure>
> (((double double) square) 5)
152587890625
> (square (square (square (square 5))))
152587890625

answer shot from my screen at 7-point font:

> (/ (log (((double (double double)) square) 5)) (log 10))
45.807: decimal digits

> (define (compose f g) (lambda (x) (f (g x))))
> (define (cube x) (* x x x))
> ((compose cube square) 5)
15625
> ((compose square cube) 5)
15625
> (define (add2 n) (+ 2 n))
> ((compose add2 square) 5)
27
> ((compose square add2) 5)
49

Draw a picture

> (define (square x) (* x x))