Lambda in different languages

- A simple example of a higher order function in Racket

1

```
(define (f g) (g 2 3))
(define (square n) (* n n))
(test (f +) 5)
(test (f *) 6)
(test (f (lambda (x y) (+ (square x)
                          (square y))))
      13)
```

- The same example can be written in JavaScript like this:

2

```
function f(g) { return g(2,3); }
function square(x) { return x*x; }
console.log(f(function (x,y)
        { return square(x) + square(y); }));
```

- In Perl:

3

```
sub f { my ($g) = @_; return $g->(2,3); }
sub square { my ($x) = @_; return $x * $x; }
print f(sub { my ($x, $y) = @_;
              return square($x) + square($y);
                 });
```

- In Ruby:

4

```
def f(g) g.call(2,3) end
def square(x) x*x end
puts f(lambda{|x,y| square(x) + square(y)});
```

- In Java:

5

```
static int f(ToIntBiFunction<Integer, Integer>
    g)
{
    return g.applyAsInt(2, 3);
}
static int square(int x) { return x*x; }
public static void main(String [] args) {
    System.out.println(f( (x,y) ->
        square(x)+square(y)));
}
```

Using Functions as Objects

- Functions as objects with one method:

```
6   (define (f x) (lambda () x))

    (define a (f 2))
    (test (a) 2)

    (define b (f 3))
    (test  (b) 3)
```

- Why stop at one level?

7

```
(define aa (f a))
(aa)     ;  #<procedure> (this is a)
((aa))   ;  2
```

- Making pairs out of functions

8

```
(define (_cons x y)
  (lambda (b)
    (if b x y)))
(define (_first x) (x #t))
(define (_rest x) (x #f))
(define a (_cons 1 'alpha))
(define b (_cons 'beta 4))

(test (_first a) 1)
(test (_rest b) 4)
(test (_rest a) 'alpha)
```

- We can replace the if with more function shenanigans:

9

```
(define (_cons x y) (lambda (s) (s x y)))
(define (_first x) (x (lambda (x y) x)))
(define (_rest x) (x (lambda (x y) y)))
(define a (_cons 1 'alpha))
(define b (_cons 'beta 4))

(test (_first a) 1)
(test (_rest b) 4)
(test (_rest a) 'alpha)
```

- Using our new "data structures":

10

```
(define numlst (_cons 1
                      (_cons  2
                              (_cons 3 empty))))

(define (sum lst)
  (cond
    [(_empty? lst) 0]
    [else (+ (_first lst) (sum (_rest lst)))]))

(test (sum numlst) 6)
```

- Giving types for these functions is a challenge.

11

```
(define (_cons [x : 'a] [y : 'b])
                : (('a 'b -> 'c) -> 'c)
  (lambda (s)
    (s x y)))
(define (_first x) (x (lambda (x y) x)))
(define (_rest x) (x (lambda (x y) y)))
(define (_empty? lst) (eq? lst empty))

(define lst (_cons 1 (_cons 2 empty)))
(test (_first lst) 1)
;(test (_first (_rest lst)) 2)
;(test (_empty? lst) #f)
```

- Finally in JavaScript:

12

```
function _cons(x,y) {
  return function(s) { return s(x, y); }
}
function _first(x) {
    return x(function(x,y){ return x; }); }
function _rest(x) {
    return x(function(x,y){ return y; }); }
a = _cons(1,2);
b = _cons(3,4);
console.log('a=<'+_first(a)+','+_rest(a)+'>');
console.log('b=<'+_first(b)+','+ _rest(b)+'>');
```

- Finally in JavaScript:

```
function _cons(x,y) {
  return function(s) { return s(x, y); }
}
function _first(x) {
    return x(function(x,y){ return x; }); }
function _rest(x) {
    return x(function(x,y){ return y; }); }
a = _cons(1,2);
b = _cons(3,4);
console.log('a=<'+_first(a)+','+_rest(a)+'>');
console.log('b=<'+_first(b)+','+ _rest(b)+'>');
```

1. So there might be a reason the type system includes a listof primitive

Currying

- A "curried" function is a function that accepts one argument and returns a function that accepts the rest.
- common with H.O. functions like map, where we want to 'fix' one argument.

13

```
(define (plus [x : Number]) : (Number ->
    Number)
  (lambda (y)
    (+ x y)))

(map (plus 1) (list 1 2 3))
```

- It's easy to write functions for translating between normal and curried versions.

14

```
(define (currify f)
  (lambda (x) (lambda (y) (f x y))))

(define plus (currify +))

(test ((plus 1) 2)  3)
(test (map (plus 1) (list 1 2 3)) (list 2 3 4))
(test (map ((currify +) 1) (list 1 2 3))
      (list 2 3 4))
```

When dealing with such higher-order code, the types are very helpful, since every arrow corresponds to a function:

```
15  (has-type currify : (('a 'b -> 'c) ->
                          ('a -> ('b -> 'c))))
```

- there are some situations where we can essentially memoize when using curried functions.
- Suppose we want a function that receives two inputs x,y and returns fib(x)*y. We use a slow fib, to make a point.

16

```
(define (fib n)
  (if (<= n 1)
    n
    (+ (fib (- n 1)) (fib (- n 2)))))
```

The function we want is:

17
```
(define (bogus x y)
  (* (fib x) y))
```

If we currify it as usual, we get:

18
```
(define (bogus x)
  (lambda (y)
    (* (fib x) y)))
```

And try this several times:

19

```
(define bogus30 (bogus 30))
(time (map bogus30 (build-list 40 (lambda (x) x))))
```

But in the definition of ‘bogus’, notice that ‘(fib x)’ does not depend on ‘y’ – so we can rewrite it a little differently:

20

```
(define (bogus x)
  (let ([fibx (fib x)])
    (lambda (y)
      (* fibx y))))
```

and trying the above again is much faster now:

21
```
(define bogus30 (bogus 30))
(time (map bogus30 (build-list 40 (lambda (x) x))))
```

Implementing Lexical Scope using Racket Closures and Environments

An alternative representation for an environment

- We've already seen how first-class functions can be used to implement "objects" that contain some information.
- We can use the same idea to represent an environment.
- The basic intuition is – an environment is a **mapping** (a function) between an identifier and some value.

- If we know all the values in advance, it's a simple case statement.

22

```
(define (my-map id)
  (case id
    [(a) 1]
    [(b) 2]
    [else (error 'my-map "free variable")]))
```

- If can define 'EmptyEnv', 'Extend', and 'lookup' with the same type signature, we can just plug them in
- For convenience, we define

23

```
(define-type-alias ENV (symbol -> VAL))
```

- Here is the previous definition of lookup

24

```
(define (lookup name env)
  (type-case ENV env
    [(EmptyEnv)
               (error 'lookup "free variable")]
    [(Extend id val rest-env)
     (if (eq? id name)
         val
         (lookup name rest-env))]))
```

- The new ENV is a function, that does the lookup itself.

25
```
(define (lookup name env)
  (env name))
```

- This suggests relocating our error message

26
```
(define (EmptyEnv)
  (lambda (id) (error 'lookup "free
     variable")))
```

- Finally, ‘Extend’ – this was previously created by the variant case of the ENV type definition:

27
```
[Extend (name : symbol) (val : VAL)
        (rest : ENV)])
```

- How do we extend a given environment? the result should be mapping

28
```
(define (Extend [id : symbol] [val : VAL]
                (rest-env : ENV)) : ENV
  (lambda (name)
    ...))
```

- if we look for ‘id’ then the result should be ‘val’:

29
```
(define (Extend id val rest-env)
  (lambda (name)
    (if (eq? name id)
      val
      ...)))
```

- Otherwise we should fall back on the remaining environment

30

```
(define (Extend id v rest-env)
  (lambda (name)
    (if (eq? name id)
      v
      (rest-env name))))
      ;; equivalent to (lookup name rest-env)
```

- Our original example of

31

```
(define (my-map id)
  (case id
    [(a) 1]
    [(b) 2]
    [else (error 'my-map "wat?")]))
```

Can be written as


```
(Extend 'a 1 (Extend 'b 2 (EmptyEnv)))
```

The new code is now the same, except for the environment code:

33

```
(test (run `{call {with {x 3}
                    {fun {y} {+ x y}}}
                  4})
      7)
(test (run `{call {call {fun {x} {call x 1}}
                        {fun {x} {fun {y} {+ x
                           y}}}}
                  123})
      124)
```