Building objects and lists from closures

- "A Standard Model for Objects" PLAI3

Functions as objects with one method:

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

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

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

Making pairs out of functions (object style)

OC

```
(define (_cons x y)
  (lambda (selector)
    (case selector
      [(first) x]
      [(rest) y])))
(define (_first o) (o 'first))
(define (_rest o) (o 'rest))
(define a (_cons 1 'alpha))
(define b (_cons 'beta 4))

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

Making pairs out of functions

```
(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:

```
(define (_cons x y) (lambda (s) (s x y)))
(define (_first pair) (pair
                        (lambda (x y) x)))
(define (_rest pair) (pair
                       (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”

```
(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.

```
(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)
```

Giving types for these functions is a challenge.

```
(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)
```

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

Lists made of functions in JavaScript

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

Implementing Lexical Scope using Racket Closures and Environments

- 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")]))
```

Or we can unroll it.

23
```
(define (my-map id)
  (if (eq? id 'a)
      1
      (if (eq? id 'b)
          2
          (error 'my-map
                 "free variable"))))
```

Try to cut down on repetition defining a local function:

```
(define (my-map id)
  (let ([extend (λ (name val thunk)
                  (if (eq? name id) val (thunk)))])
    (extend 'a 1
            (λ ()
              (extend 'b 2
                      (λ () (error 'my-map
                                   "free
                                      variable")))))))
```

This isn't obviously better than the ifs, but it does suggest a way to build up such mapping functions dynamically, by defining Extend to return a lambda.

More uses for Closures

A silly little expression language, has functions but no identifiers.

++

```
(define (++ n)
  (lambda (m)
    (+ n m)))

((++ 7)  ((++ 3)  4))
```

```
(define-type ADEX
  (Adder [n : Number])
  (Call [adder : ADEX] [arg : ADEX])
  (Num [n : Number]))
```

adex

```
(define (eval expr)
  (type-case ADEX expr
    [(Num n) expr]
    [(Adder n) expr]
    [(Call adder arg)
     (let ([addex (eval adder)]
           [argex (eval arg)])
       (Num (+ (Adder-n addex)
               (Num-n argex))))]))

(test (eval (Call (Adder 7)
                  (Call (Adder 4)
                        (Num 3))))
      (Num 14))
```

Move part of the evaluator into the function value

adex2
```
(define (eval expr)
  (type-case ADEX expr
    [(Num n) n]
    [(Adder n)
     (lambda (arg)
       (+ n arg))]
    [(Call adder arg)
     (let ([addex (eval adder)]
           [argex (eval arg)])
       (addex argex))]))

(test (eval (Call (Adder 7)
                  (Call (Adder 4)
                        (Num 3))))
      14)
```

How meta is your evaluator?

- A syntactic evaluator implements all target language behavior explicitly.
- A meta evaluator is an evaluator that uses language features of the host language to directly implement behavior of the evaluated language.

- our substitution-based FLANG evaluator was **close** to being a syntactic evaluator
- All of our evaluators rely on e.g. Racket arithmetic

- meta evaluators are easy **exactly** when there is a close match between host and target language.
- We can make our evaluator a meta evaluator by removing the encapsulation of FLANG values in a VAL type.
- This is so close to Racket, we can say something stronger.
- A meta-circular evaluator is a meta evaluator in which the implementation and the evaluated languages are the same.

└─Influence of implementation languages

- meta evaluators are easy **exactly** when there is a close match between host and target language.
- We can make our evaluator a meta evaluator by removing the encapsulation of FLANG values in a VAL type.
- This is so close to Racket, we can say something stronger.
- A meta-circular evaluator is a meta evaluator in which the implementation and the evaluated languages are the same.

1. Put differently, the trivial nature of the evaluator clues us in to the deep connection between the two languages, whatever their syntactic differences may be.

Feature Embedding

We saw that the difference between lazy evaluation and eager evaluation is in the evaluation rules for ‘let1’ forms, function applications, etc...

Eager:

```
eval({let1 {x E1} E2}) = eval(E2[eval(E1)/x])
```

Lazy:

```
eval({let1 {x E1} E2}) = eval(E2[E1/x])
```

- the first rule is eager because of we understand the mathematical notation to be eager.

Inherited laziness

Similarly, when plait args are evaluated lazily, this is a lazy evaluator (we just need to change #lang line).

lf

```
(define (eval expr)
  (type-case FLANG expr
    ⋮
    [(Let1 bound-id named-expr bound-body)
     (eval (subst bound-body
                  bound-id
                  (Num (eval named-expr))))]
      ⋮
    ))
```

- A general phenomena where some of the semantic features of the host language/notation we use gets **embedded** into the language we implement.
- Consider the code that implements arithmetic:

```
;; reducing FLANG expressions to numbers
(define (eval expr)
  (type-case FLANG expr
    [(Num n) n]
    [(Add l r) (+ (eval l) (eval r))]
    ...))
```

- What if it was written like this, would it still implement unlimited integers and exact fractions?

```
FLANG eval(FLANG expr) {
  if (is_Num(expr))
    return num_of_Num(expr);
  else if (is_Add(expr))
    return eval(lhs_of_Add(expr)) +
       eval(rhs_of_Add(expr));
  else if ...
  ...
}
```

- The bottom line is that we should be aware of “inherited” features (or lack thereof), and be very careful when we talk about semantics.
- Even the mathematical language that we use to communicate (semi-formal logic) can mean different things.

- The bottom line is that we should be aware of “inherited” features (or lack thereof), and be very careful when we talk about semantics.
- Even the mathematical language that we use to communicate (semi-formal logic) can mean different things.

1. Aside: read ”Reflections on Trusting Trust” by Ken Thompson (You can skip to the ”Stage II” part to get to the interesting stuff.)