CS301
Session 5

Where are we?

- We've added S-expressions (and booleans) to Impcore
- We've seen how to use them to implement
  - sets
  - dictionaries
- We've introduced lambda and the idea of first-class functions

Where are we going?

- A look at how functions can be treated as values
- A long look at how to exploit first-class functions

Lambda

- Creates unnamed function

```
(lambda (x) (* x 3))
```

- ... the function that multiplies its argument by 3
- In

```
(lambda (x) (+ x y))
```

- ... `x` is *bound,* but `y` is *free*

Uses of lambda

- Define nested functions using letrec
- Pass functions as parameters
- Return functions as results
- Store them in data structures

Nested functions for mutual recursion

- From problem set 2:

```
(define preord (t)
  (letrec
    ((pre* (lambda (ts)
             (if (null? ts '()
                 (append (preord (car ts))
                         (pre* (cdr ts))))))
    )
    (if (leaf? t) (list1 t)
        (cons (label t) (pre* (cdr t))))))
```


Nested functions

- Free variables

```
(define contig-sublist? (l1 l2)
  (letrec
      ((prefix
        (lambda (f s)
          (if (null? f) #t
              (if (null? s) #f
                  (if (equal? (car f) (car s))
                      (prefix (cdr f) (cdr s))
                      (prefix l1 (cdr s))))))))
    (prefix l1 l2)))
```

- Free variable `l1` used to "reset" the search!

Implementing nesting

- Easy: keep stack of "displays" at run time
  - *not* the call stack!
  - location of free variable known at compile time
- Familiar "static scoping" rule

Functions as arguments

```
(define twice (f x) (f (f x)))

(define nsq*m(n m)
  (twice (lambda (x) (* n x))
         m))
-> (nsq*m 3 2)
18
```

What are the bound and free variables?

Implementation

- Still straightforward: `n` and `m` are on the call stack while the lambda is being called in `twice`
- At runtime, maintain links backward in the call stack to find the values of free variables

Lambda: the great escape

- Things are different when we return functions as results or store them in data structures!
- The free variables "escape" their original environment
- Now we need closures: in our interpreters a lambda evaluates to a pair containing the code and the current environment

Example

```
-> (define mult-by (n) (lambda (x) (* n x)))
mult-by
-> ((mult-by 3) 4)
12
```

- Question: after `(mult-by 3)` returns, what does `n` mean?
- In a naive implementation the parameter context is gone!
- ...thus closures!

Application: Function composition

```
(define o (f g)
  (lambda (x) (f (g x))))

-> (val caddr (o car (o cdr  cdr)))
<procedure>
-> (caddr '(a b c d))
c
```


Application: currying

```
-> (define put-head (x) ((curry cons) x))
put-head
-> (val put-a (put-head 'a))
<procedure>
-> (put-a '(b c d))
(a b c d)
-> (put-a '(b l e))
(a b l e)
```


"own variables"

```
(define lock-box (key open)
  (lambda (k)
    (if (equal? k key)
        (if open
            (begin (set open #f) '(you have locked the box))
            (begin (set open #t) '(you have unlocked the box)))
        '(that is not the right key))))
-> (val box1 (lock-box 42 #f))
<procedure>
-> (val box2 (lock-box 7 #t))
<procedure>
-> (box1 42)
(you have unlocked the box)
-> (box1 42)
(you have locked the box)
-> (box2 42)
(that is not the right key)
-> (box2 7)
(you have locked the box
-> (box2 7)
(you have unlocked the box)
```


H-O functions in the standard basis

- Besides compose and curry...
- Applying predicates to lists via filter, exists?, all?
- Transforming lists via map
- General list *catamorphisms* foldl and foldr

Predicates

- Predicate: a function that returns a boolean

```
(val pos ((curry <) 0))
-> (filter pos '(-3 3 -4 4 -1 0 5))
(3 4 5)
```

- Let's implement filter on the board

The list transformer map

- Apply a function of one argument to each element

```
-> (map pos '(-3 3 -4 4 -1 0 5))
(#f #t #f #t #f #f #t)

-> (map
   (lambda (x)
    (cons x '()))
   '(a b c))
((a) (b) (c))
```


Fold operators

- As a result of

```
(foldr f z '(1 2 3))
```

- ... the input

```
(cons 1 (cons 2 (cons 3 '())))
```

- ... is transformed into the value of

```
(f    1 (f    2 (f    3 z  )))
```


Using fold

```
(define countall (x xs)
  (foldl
   (lambda (y sum)
     (+ sum (if (equal? x y)
                1
                (if (pair? y)
                    (countall x y)
                    0))))
   0 xs))
```

Using fold

```
(define preord (t)
  (if (leaf? t)
      (list1 t)
      (cons (label t)
            (foldr
             (lambda (t l)
               (append (preord t) l))
             '()
             (cdr t)))))
```


Folding from the left

- As a result of

```
(foldl f z '(1 2 3))
```

- ... the input

```
(cons 1 (cons 2 (cons 3 '())))
```

- ... is transformed into the value of

```
(f    3 (f    2 (f    1 z  )))
```


foldl versus foldr

```
-> (foldr cons '() '(1 2 3))
(1 2 3)

-> (foldl cons '() '(1 2 3))
(3 2 1)
```


Next time

- Using higher-order functions
  - polymorphism
  - continuations

CS301
Session 6

Where are we?

- We've seen how Scheme's first-class functions can be used
  - for local function definitions
  - to pass functions as parameters
  - to return functions as results
- We've looked at the standard basis functions
  `o curry all? exists? filter map foldl foldr`

Where are we going?

- Higher-order functions for polymorphism
- Higher-order functions and continuation-passing style

Polymorphism - the problem

- Sets, yes, but sets of what?

```
-> (val emptyset '())
-> (define member? (x s)
                    (exists? ((curry equal?) x) s))
-> (define add-element (x s)
                   (if (member? x s) s (cons x s)))
-> (define union (s1 s2) (foldl add-element s1 s2))
-> (define set-from-list (l)
                         (foldl add-element '() l))
-> (union '(1 2 3 4) '(2 4 6 8))
(8 6 1 2 3 4)
```

Wanted: generality

- What if we want sets of a-lists?
- The predicate `equal?` isn't the right thing!

```
(equal? '((U Thant)(I Ching)(E coli))
        '((E coli)(I Ching)(U Thant)))
#f
```

- ...but that's what's used in `member?`

Equality for a-lists

- The right test:

```
(define sub-alist? (d1 d2)
  (all?
   (lambda (pair)
     (equal? (cadr pair) (find (car pair) d2)))
   d1))

(define =alist? (d1 d2)
      (if (sub-alist? d1 d2)
          (sub-alist? d2 d1)
          #f))
```


Clunky polymorphism

- Redefine set ops to expect equality test as parameter

```
(define member? (x s eqfun)
  (exists? ((curry eqfun) x) s))
(define add-element (x s eqfun)
  (if (member? x s eqfun) s (cons x s)))
...
```

- But then we have to redefine all the set ops! And to use:

```
(member x s =alist)
(member y s' equal?)
```


Critique

- We have to pass the equality predicate around wherever we use the set - wordy, awkward, and error-prone
- Better idea: make the predicate part of the set

Polymorphism version 2

- Represent set as pair of function and data (like a simple object in an OO language)

```
(define mk-set (eqfun elements)
  (cons eqfun elements))
(define eqfun-of (set) (car set))
(define elements-of (set) (cdr set))

(val emptyset (lambda (eqfun) (mk-set eqfun '())))
(define member? (x s)
  (exists? ((curry (eqfun-of s)) x) (elements-of s)))
(define add-element (x s)
  (if (member? x s) s
      (mk-set (eqfun-of s) (cons x (elements-of s)))))
```


Critique

- Now we can use the set without explicitly mentioning the equality predicate - good!
- And there's no danger of using the wrong predicate - good!
- But every set we have contains an extra cons cell - even when there are many sets and only a few different equality predicates - which might be bad!
- To avoid the cons cell, package the set ops as a closure over the predicate

Polymorphism version 3

```
(val mk-set-ops
 (lambda (eqfun)
  (list2
  (lambda (x s)
     (exists? ((curry eqfun) x) s))
  (lambda (x s)
     (if (exists? ((curry eqfun) x) s) s
         (cons x s)))))))
(val al-nullset '())
(val list-of-al-ops (mk-set-ops =alist?))
(val al-member? (car list-of-al-ops))
(val al-add-element (cadr list-of-al-ops))
```


Critique

- We got rid of the cons cell - good!
- But we might use the wrong set op for our set - bad!
- And we have to define named functions for each type of set - which we might not like.
- Later we'll study language features to give us real polymorphism directly.

Continuation passing

- Scheme's major innovation: the notion of *continuation*
- Full Scheme has a built-in construct for continuation handling: `call/cc`
- The granddaddy of throwing/catching exceptions
- In uScheme we will program this explicitly with higher-order functions

Continuations for errors

- A-list `find` confuses unbound keys with keys bound to `'()`
- Clunky solution: special return values
- Elegant solution: client of `find` passes two functions: a success continuation and a failure continuation

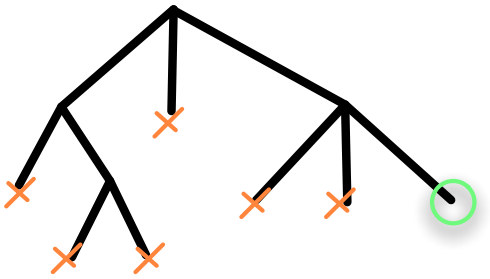
`find` with continuations

```
(define find-c (key alist success failure)
 (letrec
  ((search (lambda (alist)
     (if (null? alist)
         (failure)
         (if (equal? key (caar alist))
              (success (cadar alist))
              (search (cdr alist)))))))
    (search alist)))

(define find-default (key table default)
  (find-c key table (lambda (x) x)
    (lambda() default)))
```


Backtracking

- We can use continuations to implement a backtracking search



The SAT problem

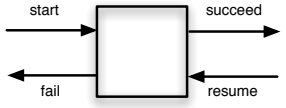
- NP-complete, but lots of practical apps
- Problem: find an assignment of booleans to variables that satisfies a CNF formula
- Example formula: $(x \vee y \vee \neg z) \wedge (w \vee y) \wedge (w \vee z)$
- Structure: conjuncts, disjuncts, literals
- Some satisfying assignments:

$$\{x \mapsto T, y \mapsto F, z \mapsto T, w \mapsto T\}$$

$$\{x \mapsto F, y \mapsto T, z \mapsto T, w \mapsto F\}$$

A basic search module

- *succeed, fail,* and *resume* are continuations



- *succeed* takes a *resume* continuation to allow backtracking in case of failure in later module
- Instantiate the module for each node of the search tree

Backtracking for SAT

- To solve a conjunction:

```
(define solve-conjunction (conjuncts cur fail succeed)
     (if (null? conjuncts)
       (succeed cur fail)
       (solve-disjunction (car conjuncts) cur fail
          (lambda (cur resume)
             (solve-conjunction (cdr conjuncts)
                                cur resume succeed)))))
```

- On backtrack, fail
- If the first conjunct is solved, continue with the rest of the conjuncts

The disjunction solver

- Now an empty list is UNsatisfiable!

```
(define solve-disjunct (disjuncts cur fail succeed)
  (if (null? disjuncts)
   (fail)
   (solve-literal (car disjuncts) cur
     (lambda ()
       (solve-disjunct (cdr disjuncts) cur fail
                       succeed))
     succeed)))
```

- One satisfiable literal is all we need
- On backtrack, look at the rest of the disjuncts

Solving a literal

- Bind if possible/needed

```
(define solve-literal (lit cur fail succeed)
  (if (satisfies? lit cur)
   (succeed cur fail)  ;fail on backtrack
   (if (binds? lit cur)
     (fail)
     (succeed (bind (variable-of lit)
                    (satisfying-value lit)) cur)
              fail))))  ;fail on backtrack
```

- No backtracking possible at this level, so we pass the failure continuation as the "resume" continuation

Watch the backtracking:

```
-> (one-solution f1)
(conjunction: ((x y) ((not x)) (y z)) soln: ())
(disjunction: (x y) soln: ())
success
(conjunction: (((not x)) (y z)) soln: ((x #t)))
(disjunction: ((not x)) soln: ((x #t)))
fail
(disjunction: () soln: ((x #t)))
fail
(disjunction: (y) soln: ())
success
(conjunction: (((not x)) (y z)) soln: ((y #t)))
(disjunction: ((not x)) soln: ((y #t)))
success
(conjunction: ((y z)) soln: ((y #t) (x #f)))
(disjunction: (y z) soln: ((y #t) (x #f)))
success
(conjunction: () soln: ((y #t) (x #f)))
((y #t) (x #f))
```


So what?

- Treating functions as first-class values is a simple idea with far-reaching consequences
- Programs can create functions on the fly, capturing the current environment in closures
- When you have a language with this feature, use functions freely for control structure, data encapsulation, and whatever else you might dream up

What next?

- An interpreter for uScheme written in ML
- ... so next week's mission is to acquire a rudimentary knowledge of ML