The current topic: Scheme

- ✓ Introduction
- ✓ Object-oriented programming: Python
- Functional programming: Scheme
  - ✓ Introduction
  - ✓ Numeric operators, REPL, quotes, functions, conditionals
  - ✓ Function examples, helper functions, let, let*
  - ✓ More function examples, higher-order functions
  - ✓ More higher-order functions, trees
  - Next up: More trees, lambda reductions, mutual recursion, examples, letrec
- Python GUI programming (Tkinter)
- Types and values
- Logic programming: Prolog
- Syntax and semantics
- Exceptions

Announcements

- Reminder: Lab 2 is due on Monday at 10:30 am.
- Term Test 2 is on November 3rd in **GB405**.
  - Aids allowed: Same as Term Test 1.
- Reminder: Deadline for Term Test 1 re-mark requests is **today**.

Review: Representing trees in Scheme

- Trees are represented as lists.
  - Each node contains its data value followed by all its children.
  - If the "child" is a "null pointer" (that is, there is no child), it is represented by the empty list.
- Example: Binary trees.

```
      4
    /   \
   2     6
        /
       5
```

```
4
 |__ 2
 |__ 6
      |__ 5
```

```
(4 (2 () ()) (6 (5 () ()) ()) )
```


Review: BST functions

- Getting the data value in a given node:

```
> (define (key node) (car node))

> (key '(4 (2 () ()) (6 (5 () ()) ()) ) )
4
```

- Getting the left subtree of a given node:

```
> (define (left node) (cadr node))

> (left '(4 (2 () ()) (6 (5 () ()) ()) ) )
(2 () ())
```

- Getting the right subtree of a given node:

```
> (define (right node) (caddr node))

> (right '(4 (2 () ()) (6 (5 () ()) ()) ) )
(6 (5 () ()) ())
```

Printing binary trees

- We want a function `print-tree` to do something like this:

```
> (print-tree '(4 (2 () (3 () ()))
                (6 (5 () ()) (7 () ()))))
4
  2
    3
  6
    5
    7

> (print-tree '(4 (2 () ()) (6 () ())))
4
  2
  6
```


Another printing example

```
> (print-tree (list2tree '(4 2 6 8 1 7)))
4
  2
    1
  6
    8
      7
```

- (In all examples, we've omitted the `#t` that `print-tree` likes to finish off with.)

Printing a binary tree

```
> (define (print-tree tree)
    (print-tree-help tree 0))

> (define (print-tree-help tree D)
    (cond ((null? tree))
          (else
            (print-spaces D)
            (display (key tree)) (newline)
            (print-tree-help (left tree) (+ D 1))
            (print-tree-help (right tree) (+ D 1)))))

> (define (print-spaces N)
    (cond ((= N 0))
          (else (display #\space) (display #\space)
                (print-spaces (- N 1)))))
```

- Note that the above code doesn't completely follow functional programming style.

Environments and local variables in lambda expressions

- Recall that function definitions are equivalent to lambda expressions:

  ```
  > (define (mult x y) (* x y))
  ```

  is equivalent to

  ```
  > (define mult (lambda (x y) (* x y)))
  ```

- Lambda expressions are a formal notation for establishing an *environment* (a local context) in which the lambda variables (the parameters to the function) are defined.

Environments and local variables in lambda expressions

- Analogy: Logic expressions also establish an environment within which variables are defined. For example:

```
∀x (P(x) ⇒ Q(x))
```

- The variable `x` in the above expression is a "bound" variable – it has meaning only within the expression. The expression establishes the environment in which `x` has meaning.
  - But the analogy isn't perfect. In a lambda expression, the variables have individual values, assigned by the caller, when the expression is evaluated.

Lambda reduction

- Lambda expressions get values by the process of *lambda reduction*: when a lambda expression is followed by a sequence of expressions, the values of those expressions are substituted for the lambda variables.

```
( (lambda (x y) (* x y)) 3 (+ 4 5))
```

⇒ [by lambda reduction] `(* 3 9)`

⇒ [by evaluation] `27`

Function calls and lambda reduction

- A function call in Scheme is really a lambda reduction:

```
> (mult 3 (+ 4 5))
```

⇒ [by evaluation]

```
( (lambda (x y) (* x y)) 3 9)
```

⇒ [by lambda reduction] `(* 3 9)`

⇒ [by evaluation] `27`

let and let* are not primitive

- That is, we can define them in terms of other forms.

```
(let ((v1 e1) … (vn en)) expr)
```

is equivalent to:

```
((lambda (v1 … vn) expr) e1 … en)
```

- For example:

```
(let ((x 5) (y 3)) (* x y))
```

is equivalent to:

```
((lambda (x y) (* x y)) 5 3)
```

let and let* are not primitive

- Similarly:

```
(let* ((v1 e1) (v2 e2)) expr)
```

is equivalent to:

```
((lambda (v1) ((lambda (v2) expr) e2)) e1)
```

- For example:

```
(let* ((x 5) (y (* x 2))) (+ x y))
```

is equivalent to:

```
((lambda (x) ((lambda (y) (+ x y)) (* x 2))) 5)
```


let and let* are not primitive

- Tracing the previous example:

```
((lambda (x) ((lambda (y) (+ x y)) (* x 2))) 5)
```

⇒ [by lambda reduction] `((lambda (y) (+ 5 y)) (* 5 2))`

⇒ [by evaluation] `((lambda (y) (+ 5 y)) 10)`

⇒ [by lambda reduction] `(+ 5 10)`

⇒ [by evaluation] `15`

- All binding of values to variables in `let` and `let*` is by parameter passing (that is, lambda reduction), *not by assignment*!

What cons *really* does

- We've been treating `cons` as a function that "appends" to the beginning of a "list".
  - This is the right general idea.
  - But it leaves out details about what's happening "behind the scenes".
- Some of you have (accidentally?) noticed what happens when the second argument given to `cons` is *not* a list:

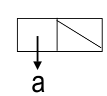
```
> (cons 'a 'b)
(a . b)

> (cons 1 2)
(1 . 2)
```

- Notice that the return values include a dot.

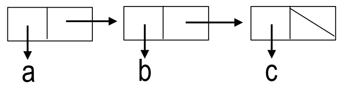
What cons *really* does

- List are implemented as linked lists.
- Each node has two parts.
  - A part storing (pointing to) data.
    - This is the node's `car`.
  - A part that's meant to store a pointer to the next node.
    - This is the node's `cdr`.
    - Think of this as the node's "next" pointer.
- `cons` creates a linked list node (also known as a *pair*).
  - The first argument to cons is stored in the new node's `car` part.
  - The second argument to cons is stored in the new node's `cdr` part.
  - e.g. `(cons 'a '())` produces:

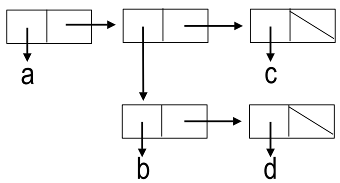


Displaying lists

- To display (output) a list, Scheme traverses the list's linked list, printing each node's `car` part.
  - For example, the following list is displayed as `(a b c)`.

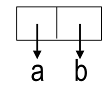


  - Observe that a properly-formed list ends with a null pointer.
  - Another example: the nested list `(a (b d) c)` is stored as:

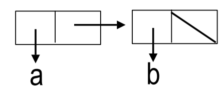


Displaying lists

- `(cons 'a 'b)` produces:



  - When traversing this list, Scheme finds that the node's `cdr` part doesn't point to the next node, but instead points to a symbol.
  - This list is displayed as `(a . b)`
  - This list's `cdr` is the atom `b`, not the list `(b)`.
  - The list `(a b)` produced by `(cons 'a '(b))` is stored as:



Mutual recursion

- *Mutual recursion* is a form of recursion where two functions call *each other* (rather than themselves).
  - Functions `f1` and `f2` are mutually recursive if `f1` calls `f2` and `f2` calls `f1`.
- Let's define variants of map that only apply the given function to certain parts of the given list (and leave other parts unchanged).
  - `map-even` takes a function `f` and a list `L`, and returns a new list in which each even-positioned element is the result of applying `f` to the corresponding element in `L`, and each odd-positioned element is simply the corresponding element in `L` unchanged.
  - `map-odd` takes a function `f` and a list `L`, and returns a new list in which each odd-positioned element is the result of applying `f` to the corresponding element in `L`, and each even-positioned element is simply the corresponding element in `L` unchanged.

map-even and map-odd

- Examples:

```
> (map-even car '((1 2 3) (4 5 6) (7 8) (a b c)))
((1 2 3) 4 (7 8) a)

> (map-odd car '((1 2 3) (4 5 6) (7 8) (a b c)))
(1 (4 5 6) 7 (a b c))

> (map-even (lambda (x) (* 2 x)) '(1 1 1 3 3 3))
(1 2 1 6 3 6)

> (map-odd (lambda (x) (* 2 x)) '(1 1 1 3 3 3))
(2 1 2 3 6 3)
```

map-even and map-odd

- We'll define `map-even` and `map-odd` so that they're mutually recursive:

```
> (define (map-odd f L)
    (cond ((null? L) ())
          (else (cons (f (car L))
                      (map-even f (cdr L))))
          ))

> (define (map-even f L)
    (cond ((null? L) ())
          (else (cons (car L)
                      (map-odd f (cdr L))))
          ))
```


map-even and map-odd

- Call: `(map-even car '((a b) (c d) (1 2) (3 4)))`

  Trace:

```
(map-even car '((a b) (c d) (1 2) (3 4)))
|(map-odd car '((c d) (1 2) (3 4)))
| (map-even car '((1 2) (3 4)))
| |(map-odd car '((3 4)))
| | (map-even car '())
| | ()
| |(3)
| ((1 2) 3)
|(c (1 2) 3)
((a b) c (1 2) 3)
```


Examples

- Write a function `makeTester` that takes two unary predicates `f1` and `f2` (a predicate is a function that returns true or false), and returns a function that takes a list and returns true iff all odd-positioned elements satisfy `f1` and all even-positioned elements satisfy `f2`. For example:

```
> ((makeTester list? symbol?) '((a b) a (c) d))
#t

> ((makeTester symbol? number?) '(a 1 2 a))
#f
```


Examples

- Defining `makeTester`, first solution:

```
> (define (makeTester f1 f2)
     (lambda (L)
       (cond ((null? L) #t)
             ((f1 (car L))
              ((makeTester f2 f1) (cdr L)))
             (else #f)
             )))
```

- This works, but notice that the function that's returned by `makeTester` calls `makeTester` each time it's called.

- Let's modify `makeTester` so the returned function does not call `makeTester`.

Examples

- Defining `makeTester`, second solution (using `map-odd` and `map-even`):

```
> (define (makeTester f1 f2)
     (lambda (L)
       (eval (cons 'and
                   (map-even f2 (map-odd f1 L))))
       ))
```

- Observe we use `map-odd` to check if the odd-positioned elements satisfy `f1`, we use `map-even` to check if the even-positioned elements satisfy `f2`, and we use `and` to combine all the results.

Examples

- Now let's try to define `makeTester` using mutually recursive lambda expressions.

```
> (define (makeTester f1 f2)
     (let ((test-odd (lambda (L)
                       (cond ((null? L) #t)
                             ((f1 (car L))
                              (test-even (cdr L)))
                             (else #f)
                             )))
           (test-even (lambda (L)
                        (cond ((null? L) #t)
                              ((f2 (car L))
                               (test-odd (cdr L)))
                              (else #f)
                              ))))
       test-odd))
```


Examples

- General idea:
  - `test-odd` checks if odd-positioned elements satisfy `f1`.
  - `test-even` checks if even-positioned elements satisfy `f2`.
  - `test-odd` and `test-even` take turns doing the checking.
    - This is accomplished using mutual recursion.
- But the code doesn't work:

```
> ((makeTester symbol? number?) '(a 1 2 a))
reference to undefined identifier: test-even
```

- What's going on?
  - The definition of `test-odd` refers to `test-even`, but we're using `let`, so the name `test-even` doesn't "exist" within the definition of `test-odd`.
  - Using `let*` instead of `let` won't solve the problem, since then `test-odd` exists within the definition of `test-even`, but `test-even` still doesn't exist within the definition of `test-odd`.

letrec

- Solution: Use `letrec`, which allows lambda expressions to refer to each other (which allows for mutual recursion).

```
> (define (makeTester f1 f2)
     (letrec ((test-odd (lambda (L)
                          (cond ((null? L) #t)
                                ((f1 (car L))
                                 (test-even (cdr L)))
                                (else #f)
                                )))
              (test-even (lambda (L)
                           (cond ((null? L) #t)
                                 ((f2 (car L))
                                  (test-odd (cdr L)))
                                 (else #f)
                                 ))))
       test-odd))
```

Examples

- Write a function `findSequence` that takes two unary predicates `f1` and `f2`, and returns a function that takes a list and returns the leftmost pair of adjacent elements in the list such that the first element of the pair satisfies `f1` and the second element satisfies `f2`, if such a pair exists, and returns `#f` otherwise. For example:

```
> ((findSequence list? symbol?) '(1 (a b) a (c) d))
((a b) a)

> ((findSequence symbol? number?) '((z) 1 a 3 2 a))
(a 3)

> ((findSequence symbol? number?) '((z) 1 a (d) 2 3))
#f
```


Examples

- Defining `findSequence`:

```
> (define (findSequence f1 f2)
    (letrec ((g (lambda (L)
                  (cond ((null? L) #f)
                        ((null? (cdr L)) #f)
                        ((and (f1 (car L))
                              (f2 (cadr L)))
                         (list (car L) (cadr L)))
                        (else (g (cdr L)))
                        ))))
             g))
```

- Observe that `letrec` is needed here, since otherwise function `g` won't be able to call itself (since the name `g` won't exist within its own definition).

Exercises

- Fix `print-tree` (defined in this lecture) so that it's clear from the output whether an only child is a right-child or a left-child.
  - Hint: Do something special when the left child is null but the right child is not null.

- Write a function `map-odd-even` that takes functions `f1` and `f2`, and a list `L`, and returns a new list in which each odd-positioned element is the result of applying `f1` to the corresponding element in `L`, and each even-positioned element is the result of applying `f2` to the corresponding element in `L`. Do not define any helper functions, and do not use `map-odd` or `map-even`. Examples:

```
> (map-odd-even car cdr '((a b) (1 2) (#t #f) (3) (4 5)))
(a (2) #t () 4)

> (map-odd-even list? symbol? '((a b) (a b) c d (e) f)))
(#t #f #f #t #t #t)
```


More exercises

- Write a function `make-odd-even` that takes functions `f1` and `f2`, and returns a function that takes a list returns a new list in which each odd-positioned element is the result of applying `f1` to the corresponding element in `L`, and each even-positioned element is the result of applying `f2` to the corresponding element in `L`. Do not use any helper functions. Instead, use `letrec` and mutually recursive lambda expressions. Examples:

```
> ((make-odd-even car cdr) '((a b) (1 2) (#t #f) (3) (4 5)))
(a (2) #t () 4)

> ((make-odd-even list? symbol?) '((a b) (a b) c d (e) f)))
(#t #f #f #t #t #t)
```