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
  - – Next up: More higher-order functions, trees
- Types and values
- Syntax and semantics
- Exceptions
- Logic programming: Prolog

Announcements

- Lab 2 is due on October 27th at 10:30 am.
  - – By the end of today's lecture, we will have covered all the material needed for the lab.
- Project
  - – Send me a list of group members by **today**.
- Re-mark request deadlines
  - – Lab 1: **Today**
  - – Term test 1: **Friday**

Review of map

- Recall that `map` takes a function `f` and a list `L`, and returns a new list in which each element is the result of applying `f` to the corresponding element in `L`.
  - – And `map` also can take a function `f` that takes more than one parameter. In that case, `map` must be given multiple lists (of the same length), where the number of lists given is equal to the number of parameters taken by `f`.
- Examples:

```
> (map not '(#t #f #t #t))
(#f #t #f #f)

> (map (lambda (X) (- X 2)) '(1 2 3))
(-1 0 1)

> (map min '(1 4) '(3 3) '(0 5) '(8 2))
(0 2)
```


What's wrong here?

```
> (define (atomcount s)
    (cond ((null? s) 0)
          ((atom? s) 1)
          (else (+ (map atomcount s)))))
> (atomcount '(a b))
+: expects argument of type <number>; given (1 1)
```

- Why doesn't this work?
  - – The call `(atomcount '(a b))` results in the "else" part being evaluated.
  - – `(map atomcount '(a b))` returns the list `(1 1)`.
  - – Then, `(+ (1 1))` is evaluated.
    - This is an error, since + is supposed to take numbers as parameters, not a list.
    - What we really want to do is evaluate `(+ 1 1)`.
    - Solution 1: Use `eval`.
    - Solution 2: Use `apply`.

Using eval to correct the problem

```
> (define (atomcount s)
    (cond ((null? s) 0)
          ((atom? s) 1)
          (else
            (eval
              (cons + (map atomcount s))))
          ))

> (atomcount '(a b))
2
> (atomcount '((1) (2 3 (4)) ((((5))))))
5
```

- Observe that `eval` causes its argument to be evaluated.
  - So `cons` puts the + inside the list, and the `eval` evaluates the list. For example:
    ```
    > (eval (cons + '(1 1)))
    2
    ```


Limitations of eval

- BUT `eval` only works in this definition of `atomcount` because numbers evaluate to themselves.
  - As in the REPL, an evaluation using `eval` causes each argument to get evaluated.
- More `eval` examples:

```
> (cons + '(1 2 3))
(#<primitive:+> 1 2 3)
> (eval (cons + '(1 2 3)))
6
> (eval '(+ 1 2 3))
6
> (eval '(abs -3))
3
> (eval '(+ (* 3 4) (* 6 6)))
48
```


eval: an example where quoting is a problem

```
> (append '(a) '(b))
(a b)
> (cons append '((a) (b)))
(#<primitive:append> (a) (b))
> (eval (cons append '((a) (b))))
reference to undefined identifier: a
```

- Problem: `eval` is trying to evaluate `(a)`, since to evaluate the expression `(append (a) (b))` it evaluates each argument.
- Fixing the problem: Use quotes to inhibit evaluation.

```
> (cons append '( '(a) '(b) ))
(#<primitive:append> '(a) '(b))
> (eval (cons append '( '(a) '(b) )) )
(a b)
```


Applying functions with apply

- `apply` is a built-in higher-order function.
  - Parameters: a function and a list
  - Result: the result of applying the given function to the parameters given by the list.

```
> (apply + '(1 2 3))
6
> (apply append '((a) (b)))
(a b)
> (apply car '((a b c)))
a
```

- Observe that `apply` does not evaluate the elements of its list parameter, so quotes are not needed *inside* the list.

Applying functions with apply

- We can use `apply` to fix `atomcount`:

```
> (define (atomcount s)
    (cond ((null? s) 0)
          ((atom? s) 1)
          (else
            (apply + (map atomcount s)))))

> (atomcount '(a (b) c))
3
```


reduce – another higher-order function

```
> (define (reduce op ls id)
      (if (null? ls)
          id
          (op (car ls)
              (reduce op (cdr ls) id))))
```

- The `reduce` function takes a binary operation `op` and applies it right-associatively to a list `ls` of an arbitrary number of elements.
- If the list `ls` is empty, reduce returns `id`.
- Informally, `reduce` "reduces" a list of elements to a single value.
- reduce is **not** equivalent to apply.

Tracing reduce with +

```
(reduce + '(1 2 3) 0)
|(reduce #<primitive:+> (1 2 3) 0)
| (reduce #<primitive:+> (2 3) 0)
| |(reduce #<primitive:+> (3) 0)
| | (reduce #<primitive:+> () 0)
| | 0
| |3
| 5
|6
6
```

- In this case, `(reduce + '(1 2 3) 0)` happens to have the same value as `(+ 1 2 3)`.
  - But what's really being computed is (1 + (2 + (3 + 0))).

Tracing reduce with /

```
(reduce / '(24 6 2) 1)
|(reduce #<primitive:/> (24 6 2) 1)
| (reduce #<primitive:/> (6 2) 1)
| |(reduce #<primitive:/> (2) 1)
| | (reduce #<primitive:/> () 1)
| | 1
| |2
| 3
|8
8
```

- Observe that `(reduce / '(24 6 2) 1)` does not have the same value as `(/ 24 6 2)`.
  - `(reduce / '(24 6 2) 1)` is equivalent to (24 / (6 / (2 / 1))) = 8.
  - `(/ 24 6 2)` is equivalent to ((24 / 6) / 2) = 2.

Tracing reduce with cons

```
(reduce cons '(z 6 c) ())
|(reduce #<primitive:cons> (z 6 c) ())
| (reduce #<primitive:cons> (6 c) ())
| |(reduce #<primitive:cons> (c) ())
| | (reduce #<primitive:cons> () ())
| | ()
| |(c)
| (6 c)
|(z 6 c)
(z 6 c)
```


reduce – another higher-order function

- Unlike some other Schemes, `mzscheme` has no built-in `reduce`.
- Warning: the `reduce` we defined is **not** the same as the `reduce` available in some other Schemes, such as MIT Scheme.
  - Ours is right-associative; MIT Scheme provides reduce (left-associative) and reduce-right (right-associative).
  - MIT Scheme's `reduce` treats lists of one element specially, returning that element and ignoring the `id` parameter.

reducing union

- Suppose we have `union`, which takes two lists representing sets and returns a list representing their union. For example:
```
> (union '(1 3) '(2 3 4))
(1 2 3 4)
> (apply union '((1 3) (2 3 4)))
(1 2 3 4)
```
- Now suppose we want to compute the union of more than two lists. We can try:
```
> (apply union '((1 3) (2 3) (4 5)))
procedure union: expects 2 arguments, given 3
```
- Solution: Use `reduce`.
```
> (reduce union '((1 3) (2 3) (4 5)) ())
(1 3 2 5 4)
```


reducing intersection

- Now suppose we have `intersection`, which takes two lists representing sets and returns a list representing their intersection. For example:
```
> (intersection '(1 3) '(2 3 4))
(3)
```
- And suppose we want to compute the intersection of more than two lists. Let's try to use `reduce`.
```
> (reduce intersection '((1 3) (2 3 4)) ())
()
```
- What went wrong?
  - In the example above, `reduce` starts by computing the intersection of `(2 3 4)` and `()`, which is `()`.

reducing intersection

- Question: How can we change reduce so that it works the way we want it to with `intersection`? (And so that it still works the way we want it to with `union`.)
- Answer: When the list given to `reduce` has exactly one element, just return that element.
  - This would mean that, for example,

    ```
    (reduce intersection '((2 3 4)) '())
    ```

    returns `(2 3 4)` rather than `()`.

reduce-left

- Let's write a version of `reduce` that:
  - Has the property required by the previous slide.
    - That is, when the given list has just one element, the element itself is returned.
  - And is tail-recursive.
- It's easier to write a left-associative version of `reduce` that has these properties.

```
> (define (reduce-left op ls id)
     (cond ((null? ls) id)
           ((null? (cdr ls)) (car ls))
           (else
              (reduce-left
                 op
                 (cons (op (car ls) (cadr ls)) (cddr ls))
                 id
                 ))))
```


reduce-left

- Tracing calls to `reduce-left`:

  Call: `(reduce-left intersection '((1 2 3) (5 1 2) (7 8 1)) '())`

  Trace:

```
(reduce-left intersection '((1 2 3) (5 1 2) (7 8 1)) '())
(reduce-left intersection '((1 2) (7 8 1)) '())
(reduce-left intersection '((1)) '())
(1)
```

  Call: `(reduce-left / '(24 6 2) 27)`

  Trace:

```
(reduce-left / '(24 6 2) 27)
(reduce-left / '(4 2) 27)
(reduce-left / '(2) 27)
2
```


Using map and reduce together

- Recall the function `cdrLists` defined in the Exercises of a previous lecture. This function takes a list L whose elements are themselves lists, and returns a list giving all the elements in the cdrs of these lists. For example:

```
> (cdrLists '((1 2) (3 4 5) (6 7)) )
(2 4 5 7)
```

- Using `map` and `reduce`, it's easy to write `cdrLists`.

```
> (define (cdrLists L)
     (reduce append (map cdr L) '()))
```

Representing trees

- How can we represent trees?
  - 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. In this case, there are always two subtrees:


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


BST (binary search tree) 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 () ()) ())
```


Insertion into a BST (binary search tree)

- Insert element `el` into `tree`:

```
(define (insert el tree)
  (cond ((null? tree) (list el () ()))
        ((= el (key tree))  ; el is already the root
         tree
         )
        ((< el (key tree))  ; insert into left subtree
         (list (key tree)
               (insert el (left tree))
               (right tree))
         )
        (else               ; insert into right subtree
         (list (key tree)
               (left tree)
               (insert el (right tree))
         ))))
```


BST insertion examples

```
> (insert 5 ())
(5 () ())

> (insert 1 '(5 () ()))
(5 (1 () ()) ())

> (insert 8 '(5 (1 () ()) ()))
(5 (1 () ()) (8 () ()))

> (insert 5 '(4 (2 () ()) (6 () (8 () ()))))
(4 (2 () ()) (6 (5 () ()) (8 () ())))

> (insert 9 '(4 (2 () ()) (6 () (8 () ()))))
(4 (2 () ()) (6 () (8 () (9 () ()))))
```

Building a BST from a list

- Given a list of (unsorted) numbers, we want to build a BST.

```
> (define (list2tree ls)
    (list2tree-help ls ()))

> (define (list2tree-help ls tree)
    (cond ((null? ls) tree)
          (else (list2tree-help (cdr ls)
                                (insert (car ls) tree)))))

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

- Observe that:
  - `list2tree-help` uses parameter `tree` as an accumulator.
  - `list2tree-help` is tail-recursive.

Tracing list2tree

- Call: `(list2tree '(4 2 6 8 1))`

  Trace:

```
(list2tree '(4 2 6 8 1))
(list2tree-help '(4 2 6 8 1) ())
(list2tree-help '(2 6 8 1) '(4 () ()) )
(list2tree-help '(6 8 1) '(4 (2 () ()) ()) )
(list2tree-help '(8 1) '(4 (2 () ()) (6 () ())) )
(list2tree-help '(1) '(4 (2 () ()) (6 () (8 () ()))) )
(list2tree-help '() '(4 (2 (1 () ()) ()) (6 () (8 () ()))) )
(4 (2 (1 () ()) ()) (6 () (8 () ())))
```


Exercises

- In a previous set of exercises (from Oct. 10), you wrote a function called `addSums` that takes a list L of numbers, and returns the total of all sums from 0 to each number. Rewrite `addSums` so that it does not use recursion, and instead uses `map` and `apply`.

- In a previous set of exercises (from Oct. 10), you wrote a function called `swapFirstTwo` that takes a list `L`, and swaps the first two elements of `L`. Then you wrote a function called `swapTwoInLists` that takes a list `L` whose elements are themselves lists, and returns a list of all the elements in all the lists in `L`, but with the first two elements in each list swapped. Rewrite `swapTwoInLists` so that it does not use recursion, and instead uses `map` and `reduce`.

More exercises

- Write a function called `evaluateList` that takes a list `L` of S-expressions and returns a list where each element is the result of evaluating the corresponding S-expression in `L`. Examples:

```
> (evaluateList '((+ 1 2) (car '(a b c)) (list 'a 'b)))
(3 a (a b))
> (evaluateList '((null? '()) (cdr '(a b c)) (/ 6 2)))
(#t (b c) 3)
```