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

Announcements

- Lab 1 has been marked.
  - A marking report has been emailed to your ECF address.
  - Deadline for requesting a re-mark is **Monday**.
    - Use the form provided on the course website.
- Term test 1 has been marked.
  - Handed back at the end of class today.
  - Average: 73.7%
  - Deadline for requesting a re-mark is Friday October 24th.
    - Use the form provided on the course website.
- Grades are now posted on the course website.
  - Your initial password is your student number.
  - Double-check the posted grades.

More announcements

- Project.
  - Send me an email with a list of group members by **Monday**.
- Lab 2.
  - Due October 27th.

Fibonacci numbers (again)

- Given a positive integer n, compute the n-th Fibonacci number.

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

> (fib 1)
1
> (fib 2)
1
> (fib 3)
2
> (fib 4)
3
```

- Problems:
  - Efficiency. (Why?)
  - What if we want a list of the first n Fibonacci numbers?

Fibonacci numbers

- A more efficient approach, using a helper function.

```
> (define (fib-help n f1 f2 last)
    (cond ((= n last) (+ f1 f2))
          (else (fib-help (+ n 1) (+ f1 f2) f1 last))
          ))

> (define (fib n)
    (if (= n 1) 1 (fib-help 2 1 0 n)))
```

- Note that `fib-help`'s parameter `n` keeps track of which Fibonacci number is currently being computed, and parameter `last` keeps track of which Fibonacci number that we ultimately want. `f1` is the previous Fibonacci number, and `f2` is the Fibonacci number that comes before `f1`.

- Observe that `fib-help` is tail-recursive.

Fibonacci numbers

- Tracing a call to the efficient version of `fib`:

  Call: `(fib 6)`

  Trace:

```
(fib 6)
(fib-help 2 1 0 6)
(fib-help 3 1 1 6)
(fib-help 4 2 1 6)
(fib-help 5 3 2 6)
(fib-help 6 5 3 6)
8
```


Fibonacci numbers

- Getting a list of the first `n` Fibonacci numbers (first attempt):

```
> (define (fiblist n)
    (cond ((= n 0) '())
          (else (cons (fib n) (fiblist (- n 1))))
          )
    )

> (fiblist 6)
(8 5 3 2 1 1)
```

- But this is inefficient.
  - Each call made by `fiblist` to `fib` repeats work done in the previous call.
  - Solution: Use the contents of the list as we build it up. That is, if we have a list of the first n-1 Fibonacci numbers, it should be very easy to add the n-th Fibonacci number to this list.

Fibonacci numbers

- Getting a list of the first n Fibonacci numbers (second attempt):

```
> (define (fiblist n)
    (cond ((= n 1) '(1))
          ((= n 2) '(1 1))
          (else (cons (+ (car (fiblist (- n 1)))
                         (cadr (fiblist (- n 1)))
                         )
                      (fiblist (- n 1))))
          )
    )

> (fiblist 6)
(8 5 3 2 1 1)
```

- But this is still inefficient – we're constructing the same list **three times** at each recursive step!
- We can do much better.
  - Approach 1: Using `let`.
  - Approach 2: Using a helper function.

Fibonacci numbers

- Getting a list of Fibonacci numbers (more efficient version):

```
> (define (fiblist n)
    (cond ((= n 1) '(1))
          ((= n 2) '(1 1))
          (else (let ((f (fiblist (- n 1))))
                  (cons (+ (car f) (cadr f)) f)
                  ))
          )
    )

> (fiblist 6)
(8 5 3 2 1 1)
```


Fibonacci numbers

- Getting a list of Fibonacci numbers (most efficient version):

```
> (define (fiblist-help n f last)
    (let ((new-f (cons (+ (car f) (cadr f)) f)))
      (cond ((= n last) new-f)
            (else (fiblist-help (+ n 1) new-f last))
            )))

> (define (fiblist n)
    (cond ((= n 1) '(1))
          ((= n 2) '(1 1))
          (else (fiblist-help 3 '(1 1) n))
          ))
```

- Observe that `fiblist-help` is tail-recursive, and its parameter `f` acts as an accumulator.

Fibonacci numbers

- Tracing a call to the most efficient version of `fiblist`:

Call: `(fiblist 6)`

Trace:

```
(fiblist 6)
(fiblist-help 3 '(1 1) 6)
(fiblist-help 4 '(2 1 1) 6)
(fiblist-help 5 '(3 2 1 1) 6)
(fiblist-help 6 '(5 3 2 1 1) 6)
(8 5 3 2 1 1)
```


Equality checking

- The `eq?` predicate doesn't work for lists. :

```
> (eq? (cons 'a '()) (cons 'a '()))
#f
```

- Why not?
  - The first `(cons 'a '())` makes a new list.
  - The second `(cons 'a '())` makes *another* new list.
  - `eq?` checks whether its two arguments are *the same*.
  - And they're not: they're two separate lists.

- Lists are stored as pairs of pointers: one to the first element (the car) and one to the rest of the list (the cdr).

- Symbols and numbers are stored uniquely, so `eq?` works on them.

Equality checking for lists

- For lists, we need a comparison function to check for the same *structure* in two lists. This is what the built-in function `equal?` does. Let's define our own version of `equal?`. We'll use the `atom?` function we previously defined.

```
> (define (equal? x y)
      (or (and (atom? x) (atom? y) (eq? x y))
          (and (not (atom? x))
               (not (atom? y))
               (equal? (car x) (car y))
               (equal? (cdr x) (cdr y)))))

> (equal? 'a 'a)
#t
> (equal? '(a (b)) '(a (b)))
#t
> (equal? '((a)) '(a))
#f
```


Sum of all the numbers in a list of lists

- Parameter: a nested list of numbers.
- Result: the sum of all the numbers in the parameter.

```
  >(define (sum-list-nested ls)
     (cond ((null? ls) 0)
           ((list? (car ls))
            (+ (sum-list-nested (car ls))
               (sum-list-nested (cdr ls))))
           (else (+ (car ls)
                    (sum-list-nested (cdr ls))))))

> (sum-list-nested '(1 (3 (4 5)) 5))
18
```

- This is car-cdr recursion again:
  - If the first element is a list, then recursion on car processes the nested level.
  - Then recursion on cdr advances the computation to the next element of the list.

Higher-order functions

- A <u>*higher-order function*</u> is a function that takes a function as a parameter, returns a function, or does both.

- For example, the function you'll write for Exercise 6 of Lab 2 takes a list of functions as a parameter, and returns the composition of these functions.
  - The return value is a function.
  - Let's see an example of this in math (rather than in Scheme):
    Define `f(x) = x+2`.
    Define `g(x) = 2*x`.
    Let `h = compose([f, g])`. That is, `compose` "returns" a function, which we're "assigning" to `h`.
    Then `h(x) = f(g(x))`.
    e.g. `h(3) = f(g(3)) = f(2*3) = 6+2 = 8`.

Functions as parameters

- A higher-order function that takes a function as a parameter:

```
> (define (all-num-f f lst)
    (cond ((all-num lst) (f lst))
          (else 'error)))

> (all-num-f abs-list '(1 -2 3))
(1 2 3)
> (all-num-f car '(1 -2 3))
1
> (all-num-f abs-list '(1 a))
error
```

- We assume that helper function `all-num` has been defined to return true iff its parameter is a list containing only numbers. (Exercise: write this helper function.)
- `all-num-f` returns the result of calling `f` on `lst`.

Lambda expressions

- A *lambda expression* is a function without a name.
  - Defined in terms of the action performed on a list of parameters.
  - More formally, a lambda expression *evaluates* to an unnamed function.
    - That is, the result of the expression is an unnamed function.

```
> ((lambda (x y z) (+ x y z)) 1 2 3)
6

> ((lambda (x y) (cons x y)) 1 '(a b))
(1 a b)

> ((lambda (f x) (f x)) car '(9 8 7))
9
```


Functions as return values

- A higher-order function that returns a function as its value:

```
> (define (plus-list x)
     (cond ((number? x)
             (lambda (y) (+ (sum-n x) y)))
            ((list? x)
             (lambda (y) (+ (sum-list x) y)))
            (else (lambda (x) x))
            ))

> ((plus-list 3) 4)
10
> ((plus-list '(1 3 5)) 5)
14
> ((plus-list 'a) 5)
5
```

- Recall that `(sum-n x)` returns the sum of the numbers from 0 to `x`, and `(sum-list x)` returns the sum of the numbers in list `x`.

Functions as return values

- Observe that:
  - `(plus-list 3)` is a function that takes a single parameter, and adds 6 to this parameter.
  - `(plus-list '(1 3 5))` is a function that takes a single parameter, and adds 9 to this parameter.
  - `(plus-list 'a)` is the identity function (it takes a single parameter and returns it).

map

- `map` is a built-in higher-order function.
  - Parameters: a function and a list
  - Result: a new list in which each element is the result of applying the function parameter to the corresponding element of the list parameter
- Examples:

```
> (map abs '(-1 2 -3 4))
(1 2 3 4)
> (map (lambda (x) (+ 1 x)) '(-1 2 -3))
(0 3 -2)
> (map car '((a b c) (d e f) (g h i)))
(a d g)
> (map cdr '((a b c) (d e f) (g h i) (j k l)))
((b c) (e f) (h i) (k l))
```

map

- We could define our own `map` like this:

```
> (define (map f l)
     (cond ((null? l) ())
           (else (cons (f (car l))
                       (map f (cdr l))))))
```

- Unlike ours, the built-in `map` can take more than two arguments.
  - This allows it to work with functions f that need more than one argument.
  - Examples:

```
> (map cons '(a b c) '((1) (2) (3)))
((a 1) (b 2) (c 3))
> (map + '(1 2 3) '(4 5 6) '(7 8 9))
(12 15 18)
> (map max '(1 4 8) '(2 5 2) '(9 4 1) '(0 0 0))
(9 5 8)
```


Exercises

- Write a function called `addToEnd` that takes an element `e` and a list `L`, and adds `e` to the end of `L`. Do not use recursion. Example:

```
> (addToEnd 'd '(1 2 3))
(1 2 3 d)
```

- Write a function called `funAddToEnd` that takes an element `e` and returns a function that takes a list and adds `e` to the end of the list. Example:

```
> ((funAddToEnd 'a) '(2 3 4))
(2 3 4 a)
```

- Write a function called `fixFirst` that takes a binary function `f` and a parameter `p`, and returns a function that is the same as `f` except the first parameter is fixed to be `p`. Examples:

```
> ((fixFirst cons 'z) '(a b c))
(z a b c)
> ((fixFirst append '(1 2)) '(3 4))
(1 2 3 4)
```