The current topic: Scheme

✓ Introduction
✓ Object-oriented programming: Python
• Functional programming: Scheme
  ✓ Introduction
  ✓ Numeric operators, REPL, quotes, functions, conditionals
  – Next up: Function examples, helper functions, let, let*
• Types and values
• Syntax and semantics
• Exceptions
• Logic programming: Prolog

Announcements

• Project has been posted on the course website.
  – Due November 17th.
  – Send me an email with a list of group members by October 20th.

• Lab 2 will be available soon.
  – Due October 27th.
  – Six exercises.
  – By the end of today’s class, we’ll have covered the material needed for the first
    four exercises.

• Office hours next week:
  – The office hour on Wednesday (October 15th) is cancelled. Instead, there will be
    an office hour on Thursday (October 16th), 1:00-2:00, in SF3207.

Sum to n

• Given a non-negative integer n, computer the sum of all integers from 0
  to n.

  > (define (sum-n n)
     (cond ((= n 0) 0)
           (else (+ n (sum-n (- n 1))))
     )
  )

  > (sum-n 6)
  21

• At each step, a counter is decremented.
  – Recursion is the same in Scheme as anywhere: you need a base case, and a
    recursive step that solves a smaller version of the same problem.

Factorial

• Given a non-negative integer n, compute n!.
  – Recall that 0! is defined to be 1.

  > (define (factorial n)
     (cond ((= n 0) 1)
           (else (* n (factorial (- n 1))))
     )
  )

  > (factorial 5)
  120

• As in the previous example, a counter is decremented at each step.
Length

- Given a list, compute its length. There is already a built-in \texttt{length} function that computes this. We can also define our own version of \texttt{length}:

\begin{verbatim}
> (define (length x)
  (cond ((null? x) 0)
        (else (+ 1 (length (cdr x))))
  )
)
> (length '(1 2 3)) 3
\end{verbatim}

- The recursion used in \texttt{length} is called "cdr-recursion".
  - At each step, a shorter list is passed to the next function call.

Tracing length

- Tracing (by hand) a call to \texttt{length}:

  Call: (length '(a b c))

  Trace:
  
  (length '(a b c))
  (+ 1 (length '(b c)))
  (+ 1 (+ 1 (length '(c))))
  (+ 1 (+ 1 (+ 1 (length ()))))
  (+ 1 (+ 1 (+ 1 0)))
  (+ 1 (+ 1 1))
  (+ 1 2)
  3

Absolute value of all the members

- Parameter: a list of numbers.
- Result: a list containing the absolute values of the parameter's members.
- We'll make use of the \texttt{abs-val} function we defined last class.

\begin{verbatim}
> (define (abs-list ls)
  (cond ((null? ls) ())
        (else (cons (abs-val (car ls))
                     (abs-list (cdr ls))))
  )
)
> (abs-list '(1 2 3))
(1 2 3)
\end{verbatim}

- This is another example of cdr-recursion.

Sum of the members of a list

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

\begin{verbatim}
> (define (sum-list ls)
  (cond ((null? ls) 0)
        (else (+ (car ls) (sum-list (cdr ls))))
  )
)
> (sum-list '(2 3 4))
9
\end{verbatim}

- Notice yet again the standard recursive structure:
  - The base case, which stops the recursion.
  - The recursive case, giving a smaller example of the same problem.
    - cdr recursion, passing on a shorter list
append

• append is a built-in function that, given two lists L1 and L2, returns a list formed by appending L2 to L1.

> (append '(1 2) '(3 4 5))
(1 2 3 4 5)
> (append '(1 2) '(3 (4) 5))
(1 2 3 (4) 5)
> (append '()' '(1 4 5))
(1 4 5)
> (append '(1 4 5) '())
(1 4 5)
> (append '() '())
()

We can also define our own version of append.

> (define (append list1 list2)
  (cond ((null? list1) list2)
        (else (cons (car list1)
                     (append (cdr list1) list2) )))
)

Counting atoms

• Parameter: a (possibly nested) list.
• Result: the number of atoms in the list.

> (define (atomcount x)
  (cond ((null? x) 0)
        ((atom? x) 1)
        (else (+ (atomcount (car x))
                  (atomcount (cdr x)) ))
  )
> (atomcount '(1 2))
2
> (atomcount '(1 (2 (3)) (5)))
4

This is called "car-cdr recursion".
  – We go off in two directions at once.

Tracing atomcount

> (atomcount '(1 (2 (3)) (5)))
| (atomcount '(1 (2 (3)) (5))) | 1
| (atomcount (1 (2 (3)) (5))) | 2
| (atomcount 1) | (atomcount (5))
| 1 | (atomcount 5)
| (atomcount (2 (3))) | (atomcount 5)
| (atomcount 2) | (atomcount (3))
| 1 | (atomcount 3)
| (atomcount 3) | (atomcount (5))
| (atomcount 1) | 0
| 1 | (atomcount (5))
| 1 | (atomcount (3))
| 0 | (atomcount (5))
| 3 | (atomcount (3))
| 4 | (atomcount (5))
| 4 | (atomcount (5))
| 0 | (atomcount (5))
| 1 | (atomcount (3))
| 1 | (atomcount (5))
| 0 | (atomcount (5))
| 3 | (atomcount (3))
| 4 | (atomcount (5))
| 4 | (atomcount (5))
| 0 | (atomcount (5))
| 1 | (atomcount (3))
| 1 | (atomcount (5))
| 0 | (atomcount (5))
| 3 | (atomcount (3))
| 4 | (atomcount (5))
| 4 | (atomcount (5))
| 0 | (atomcount (5))
| 1 | (atomcount (3))
| 1 | (atomcount (5))
| 0 | (atomcount (5))
| 3 | (atomcount (3))
| 4 | (atomcount (5))
| 4 | (atomcount (5))
| 0 | (atomcount (5))
| 1 | (atomcount (3))
| 1 | (atomcount (5))
| 0 | (atomcount (5))
| 3 | (atomcount (3))
| 4 | (atomcount (5))
| 4 | (atomcount (5))
| 0 | (atomcount (5))
| 1 | (atomcount (3))
| 1 | (atomcount (5))
| 0 | (atomcount (5))
| 3 | (atomcount (3))
| 4 | (atomcount (5))
| 4 | (atomcount (5))
| 0 | (atomcount (5))
| 1 | (atomcount (3))
| 1 | (atomcount (5))
| 0 | (atomcount (5))
| 3 | (atomcount (3))
| 4 | (atomcount (5))
| 4 | (atomcount (5))
| 0 | (atomcount (5))
| 1 | (atomcount (3))
| 1 | (atomcount (5))
| 0 | (atomcount (5))
| 3 | (atomcount (3))
| 4 | (atomcount (5))
| 4 | (atomcount (5))
| 0 | (atomcount (5))
| 1 | (atomcount (3))
| 1 | (atomcount (5))
| 0 | (atomcount (5))
| 3 | (atomcount (3))
| 4 | (atomcount (5))
| 4 | (atomcount (5))
| 0 | (atomcount (5))
| 1 | (atomcount (3))
| 1 | (atomcount (5))
| 0 | (atomcount (5))
| 3 | (atomcount (3))
| 4 | (atomcount (5))
| 4 | (atomcount (5))
| 0 | (atomcount (5))
| 1 | (atomcount (3))
| 1 | (atomcount (5))
| 0 | (atomcount (5))
| 3 | (atomcount (3))
| 4 | (atomcount (5))
| 4 | (atomcount (5))
| 0 | (atomcount (5))
| 1 | (atomcount (3))
| 1 | (atomcount (5))
| 0 | (atomcount (5))
| 3 | (atomcount (3))
| 4 | (atomcount (5))
| 4 | (atomcount (5))
| 0 | (atomcount (5))
| 1 | (atomcount (3))
| 1 | (atomcount (5))
| 0 | (atomcount (5))
| 3 | (atomcount (3))
| 4 | (atomcount (5))
| 4 | (atomcount (5))
| 0 | (atomcount (5))
| 1 | (atomcount (3))
| 1 | (atomcount (5))
| 0 | (atomcount (5))
| 3 | (atomcount (3))
| 4 | (atomcount (5))
| 4 | (atomcount (5))
| 0 | (atomcount (5))
| 1 | (atomcount (3))
| 1 | (atomcount (5))
| 0 | (atomcount (5))
| 3 | (atomcount (3))
| 4 | (atomcount (5))
| 4 | (atomcount (5))
| 0 | (atomcount (5))
| 1 | (atomcount (3))
| 1 | (atomcount (5))
| 0 | (atomcount (5))
| 3 | (atomcount (3))
| 4 | (atomcount (5))
| 4 | (atomcount (5))
| 0 | (atomcount (5))
| 1 | (atomcount (3))
| 1 | (atomic...
Efficiency

- A function that, given two lists, returns -1 if both lists are empty, and otherwise returns the length of the longest list:

  ```scheme```
  > (define (longest-nonzero x y)
  >   (cond ((and (null? x) (null? y)) -1)
  >           ((> (length x) (length y)) (length x))
  >           (else (length y)))
  >)
  ```

- Problem: Evaluating the same expression twice.
  - length is called on the same argument more than once.
  - We'd like to be able to reuse the result instead.

- Without an assignment statement, what can we do?

Efficiency: helper function

- One solution: Bind values to parameters in a helper function:

  ```scheme```
  > (define (maximum x y) ;; or use the built-in max function.
  >   (cond ((> x y) x)
  >          (else y)))
  > (define (longest-nonzero x y)
  >   (cond ((and (null? x) (null? y)) -1)
  >          (else (maximum (length x) (length y))))
  >)
  ```

- Observe that length is now called on each argument just once.
  - The results can be used more than once within the helper function, since they are bound to the helper function's parameters.

Efficiency: let and let*

- What if we don’t want to define a helper function? How can we still reuse the results of a function call?

- Solution: Use a `let` or `let*` construct that binds variables to expression results.

- General form:
  ```scheme```
  ```
  (let ((var1 expr1) ... (varn exprn))
    <vars are now defined and can be used here>  )
  ```
  ```
  (let* ((var1 expr1) ... (varn exprn))
    <vars are defined and can be used here>  )
  ```
  ```

- This is not the same as variable assignment, since it doesn’t let us modify the value of a variable.
  - This is just a convenient way of doing what helper functions already let us do.

Efficiency: let and let*

- What’s the difference between
  ```scheme```
  ```
  (let ((v1 e1) ... (vn en))
    expr)
  ```
  and
  ```scheme```
  ```
  (let* ((v1 e1) ... (vn en))
    expr)
  ```
  ```

- Both establish the variables v1, ..., vn to have values e1, ..., en in the expression expr.
  - let does the binding in parallel (which means the order of binding has no effect).
  - let* does the binding in sequence.
    - Earlier definitions can be used in later ones.
    - For example, you can use the value of v1 when defining v2 and v3.

```scheme```
```
let and let* examples

> (let ((x 2)) (* x x))
4

> (let ((x 4)) (let ((y (+ x 2))) (* x y)))
24

> (let ((x 4) (y (+ x 2))) (* x y))
reference to undefined identifier: x

> (let* ((x 4) (y (+ x 2))) (* x y))
24

longest-nonzero with let

> (define (longest-nonzero x y)
    (let ((lenx (length x))
          (leny (length y))
          )
      (cond ((and (= 0 lenx) (= 0 leny)) -1 )
            ((> lenx leny) lenx )
            (else leny )
            )))

• Observe that length is called on each argument just once.

• Another possible improvement:
  – Note that length gets called (twice) even when x and y are both empty.
  – It might be faster to perform a null? test first, and postpone the let definitions until after this test.

longest-nonzero yet again

(define (longest-nonzero x y)
  (if (and (null? x) (null? y))
      -1
      (let ((lenx (length x))
            (leny (length y))
            )
          (if (> lenx leny) lenx leny)
          )))

Another inefficient example

• Let’s write a function rev, to return its parameter with the elements in reverse order.
  – Note that there is already a built-in reverse function that does this.

> (define (rev lst)
    (cond ((null? lst) ())
          (else (append (rev (cdr lst))
                        (list (car lst))))))

> (rev '(1 2 3))
(3 2 1)

• It works, but there are a lot of list operations going on.
A more efficient rev, with an accumulator

> (define (rev lst) (rev-rec lst ()))

> (define (rev-rec lst acc)
   (cond ((null? lst) acc)
         (else (rev-rec (cdr lst)
                       (cons (car lst) acc))))
   )

> (rev '(a b c d))
(d c b a)

• Now each element of the original list only needs to be added to another
list once, and it goes on the front, where the work is cheap.

• Observe that rev-rec’s second parameter “accumulates” the result.

Tracing rev

• Tracing a call to rev:

   Call: (rev '(a b c d))

   Trace:
   (rev '(a b c d))
   (rev-rec '(a b c d) ())
   (rev-rec '(b c d) '(a))
   (rev-rec '(c d) '(b a))
   (rev-rec '(d) '(c b a))
   (rev-rec () '(d c b a))
   '(d c b a)

• Note that whenever rev-rec makes a recursive call, it returns whatever
the recursive call returns (there is no further computation). This is known
as tail recursion. This form of recursion can be implemented very
efficiently. Why?

Exercises

• Write a function called swapFirstTwo that takes a list L, and swaps
the first two elements of L. e.g:
  > (swapFirstTwo '(1 2 3 4))
  (2 1 3 4)

• Write 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. e.g.
  > (swapTwoInLists '((1 2 3) (4 5 6) (7 8)) )
  (2 1 3 5 4 6 8 7)

• Write a function called cdrLists that takes a list L whose elements are
themselves lists, and returns a list giving all the elements in the cdrs of
these lists. e.g:
  > (cdrLists '((1 2) (3 4 5) (6)) )
  (2 4 5)

More Exercises

• Write a function called addSums that takes a list L of numbers, and
returns the total of all sums from 0 to each number. e.g.
  > (addSums '(1 3 5)) ; this is 1 + 6 + 15
  22

• Re-write addSums so that your solution uses tail recursion. You’ll need
to write an appropriate helper function.