CSG 711 Fall, 2004
Dr. Wand Readings: EOPL, Chap. 1

Lecture 0: Introduction to Scheme

You should know most of this material before you start the course.

Basics

Interaction loop.

```
% scheme
Chez Scheme Version 6.1
Copyright (c) 1998 Cadence Research Systems

> (load "~/init.scm")
sllgen.scm 2000-09-25 11:48
define-datatype.scm version J3 2000-12-22 07:12
test-harness.scm: unified test harness for COM 3352 2001-11-08 17:14
test-suite.scm: unified test suite for COM 3352 2001-10-17 14:57
> 3                                      ; ">" indicates prompt
3

> (+ 1 3)                                 ; "(+ 1 3)" is input
4                                        ; "4" is output

(a b c)
```

The special form `(define` *name exp*`)` *binds* the variable *name* to the value of the expression *exp*.

```
(define x 3)

x

(+ 1 x)
```

```
(+ x x)

(define l (a b c))

(define u (+ 1 x))
```

Can use ordinary algebraic manipulation to reason about expressions:

```
(+ (* x x) u)
= (+ (* 3 3) 4)
= (+ 9 4)
= 13
```

Much more about this later.

Creating procedures with lambda:

```
(lambda (x) body)
```

Here `x` is the *formal parameter* or *bound variable*.

```
(lambda (x) (+ 1 x))

((lambda (x) (+ 1 x)) 4)

(define mysucc (lambda (x) (+ 1 x)))

(mysucc 4)
```

In general, we will have

```
(mysucc x) = (+ 1 x)
```

Procedures can have more than one argument:

```
(define myplus (lambda (x y) (+ x y)))

(+ 3 4)
((lambda (x y) (+ x y)) 3 4)
(myplus 3 4)
```

Procedures can take other procedures as arguments:

```
((lambda (f x) (f x 3))
 myplus 4)
```

Procedures can return other procedures; this is called Currying:

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

(define add2 (twice (lambda (x) (+ x 1))))

(define f-of-x+1
  (lambda (f)
    (lambda (x)
      (f (+ x 1)))))

So

(f-of-x+1 f) = (lambda (x) (f (+ x 1)))

and therefore

((f-of-x+1 f) x) = (f (+ x 1))


(f-of-x+1 add2)  -- takes x, sends x+1 to add2

((f-of-x+1 add2) 3) = (add2 4) = 6

(f-of-x+1 (f-of-x+1 add2))  ; a perfectly good procedure-- what does
                            ; it do?

((f-of-x+1 (f-of-x+1 add2)) 4)

= ((f-of-x+1 add2) 5)
```

```
= (add2 6)

= 8
```

Much more on this later.

List Processing

Scheme allows several different kinds of values:

Numbers integers (for now)

Procedures We've seen these already

Symbols A symbol is a sequence of letters and digits starting with a letter (can also include other characters, such as `-`, `$`, `=`, `*`, `/`, `?`, `_` (the "extended alphabetic characters"). See Scheme manual for details.)

Strings `"this is a string"` – we'll leave this until later.

Characters `#\a`, `#\b`, `#\newline`, etc. See Scheme manual for details.

Vectors Like arrays. See the Scheme manual for details.

Lists A list is a sequence $(v_0 \ \ldots \ v_{n-1})$ where each v_i is a value (either a number, procedure, symbol, string, character, vector, or list).

Examples of lists:

```
()
(turkey)
(atom)
((turkey) (atom))
xyz -- a symbol, not a list
(xyz)
(((turkey) (atom)) (xyz))
(x y z (u v blah blah) (blah blah))
((madonna "madonna" #\m #\a #\d #\o #\n #\n #\a)
 (britney "britney" #\b #\r #\i #\t #\n #\e #\y)
 (n-sync  "n'sync"  #\n #\' #\s #\y #\n #\c))
```

Putting literals in programs:

```
(quote (a b c))                         ; a list literal
'(a b c)                                ; a list literal
(a b c)                                 ; a procedure call
'()                                     ; another list literal
'a                                      ; a symbol
"a"                                     ; a string
#\a                                     ; a character
a                                       ; a variable, not a literal!
```

Basic operations on lists

`car`: if `l` is $(v_0 \ \ldots \ v_{n-1})$, then `(car l)` is v_0. The `car` of the empty list is undefined.

`e.g. (define l '(a b c))  (car l) => a`

`cdr`: if `l` is $(v_0 \ v_1 \ \ldots \ v_{n-1})$, then `(cdr l)` is $(v_1 \ \ldots \ v_{n-1})$. The `cdr` of the empty list is undefined.

`(cdr l) => (b c)`

Examples of car and cdr on previous lists.

Combining car and cdr:

```
(car (cdr l))
(cdr (cdr l))
etc.
```

```
(define first car)
(define second (lambda (l) (car (cdr l))))
```

etc.

Building lists

`cons`: if `v` is any value v, and `l` is the list `(`$v_0 \ \ldots \ v_{n-1}$`)`, then `(cons v l)` is the list `(`$v \ v_0 \ldots \ v_{n-1}$`)`

`(cons v l)` builds a list whose car is v and whose cdr is l. Can think of `cons` as a prefix operator.

Examples:

```
(cons 'a  '(b c d))  => (a b c d)

(cons '(a b) '(c d)) => ((a b) c d)

(cons '(a b) '((foo bar) baz)) => ((a b) (foo bar) baz)

(car (cons v l)) = v
(cdr (cons v l)) = l
```

Can write the types of cons, car, and cdr as follows:

```
cons : value * list -> list
car  : list -> value
cdr  : list -> list
```

Other ways of building lists

`list` takes an arbitrary number of arguments and produces a list containing the values of those arguments, eg

```
if lst1 is (foo bar), then
(list 'a 3 lst1 '(7)) => (a 3 (foo bar) (7))
```

Used only occasionally, when you know exactly what the elements of the list you want to build look like. For us, this will be pretty rare.

`append` takes two lists and produces a list consisting of the elements of the first list followed by the elements of the second list, eg:

```
if lst1 is (foo bar) then
(append lst1 '(a b)) => (foo bar a b)
```

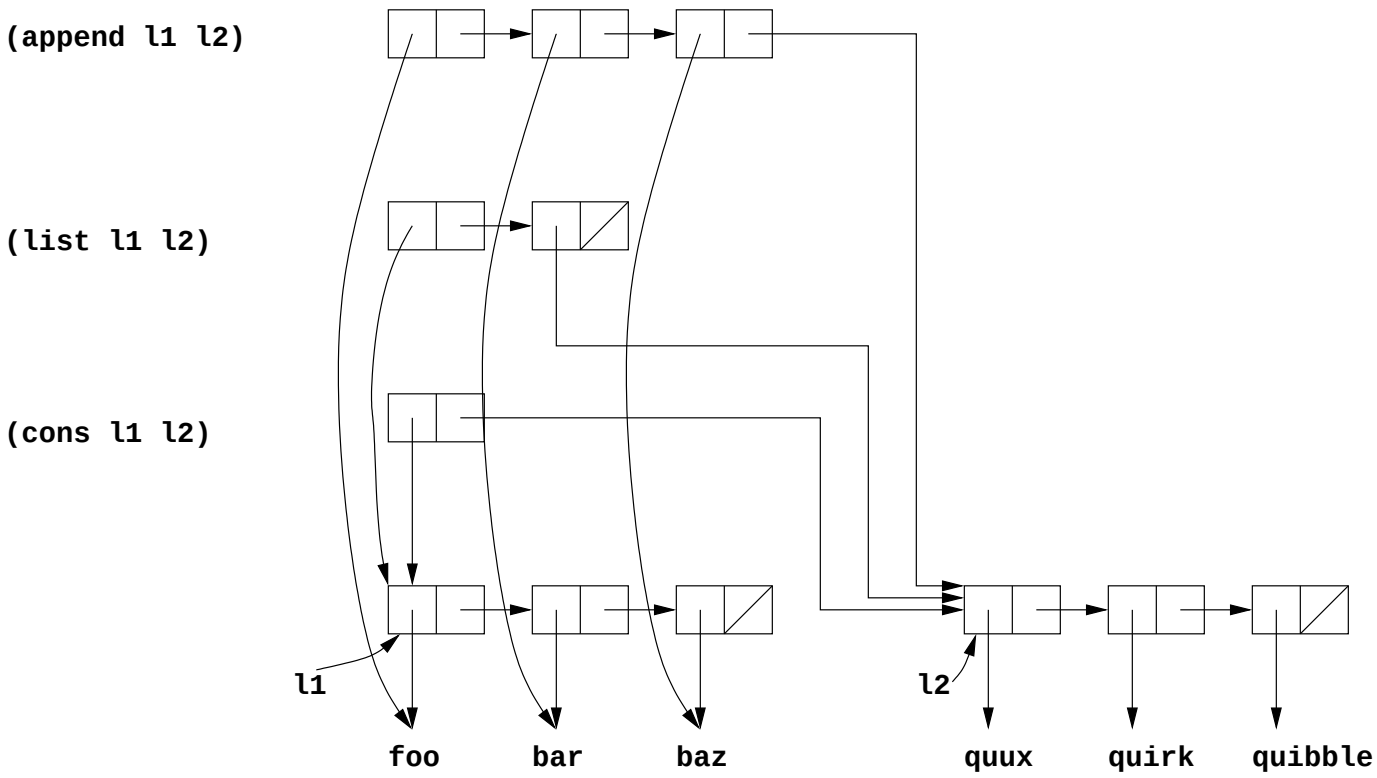
Comparing `cons`, `list`, and `append`:

```
if lst1 is (foo bar) and lst2 is (baz quux), then

(cons lst1 lst2) => ((foo bar) baz quux)
(list lst1 lst2) => ((foo bar) (baz quux))
(append lst1 lst2) => (foo bar baz quux)
```

In general if $l1$ has n elements and $l2$ has m elements, then:

`(cons` $l1$ $l2$`)` has $m+1$ elements.
`(list` $l1$ $l2$`)` has 2 elements.
`(append` $l1$ $l2$`)` has $n+m$ elements.



This is important– study this until you understand it.

Booleans

Literals:

`#t, #f`

Predicates:

```
(number? v)
(symbol? v)
(string? v)
(null? v)
(pair? v)
(eqv? s1 s2)     --works on symbols
(= n1 n2)       --works on numbers
(zero? n)
(> n1 n2)
etc
```

Conditional:

```
(if bool e1 e2)
```

```
(lambda (x)
  (if (> x 0)                           ; the test _guards_ the division
    (/ y x)
    (error 'sample "Airspeed shouldn't be 0")))
```

```
(lambda (x)                             ; x is a possibly-empty list
  (if (null? x)                         ; the test _guards_ the car
                                        ; and cdr
    (...)
    (... (car x) ... (cdr x) ...)))
```

Defining Procedures Recursively

Imagine we want to define a function to find powers, eg. $e(n,x) = x^n$.

It is easy to define a bunch of functions $e_0(x) = x^0$, $e_1(x) = x^1$, $e_2(x) = x^2$:

$$\begin{aligned} e_0(x) &= 1 \\ e_1(x) &= x \times e_0(x) \\ e_2(x) &= x \times e_1(x) \\ e_3(x) &= x \times e_2(x) \end{aligned}$$

and in general, we can define, for $n > 0$,

$$e_n(x) = x \times e_{n-1}(x)$$

How did we do this? At each stage, we used the fact that we had already solved the problem for smaller n. This is the principle of mathematical induction.

How can we parameterize this construction to get a single procedure e?

If `n` is 0, we know the answer: `(e 0 x) = 1`.

If n is greater than 0, assume we know how to solve the problem for $n-1$. Then the answer is `(e n x) = (* x (e (- n 1) x))`.

```
(define e
  (lambda (n x)
    (if (zero? n)
        1
        (* x
           (e (- n 1) x)))))
```

Why does this work? Let's prove it works for any n, by induction on n:

(1) It surely works for n=0.

(2) Now assume (for the moment) that it works when `n` $= k$. Then it works when `n` $= k+1$. Why? Because then (e n x) = (* x (e k x)), and we know e works when its first argument is k. So it gives the right answer when its first argument is $k+1$.

What's the moral? If we can reduce the problem to a smaller subproblem, then we can call the procedure itself ("recursively") to solve the smaller subproblem. Then, as we call the procedure, we ask it to work on smaller and smaller subproblems, so eventually we will ask it about something that it can solve directly (eg n=0, the basis step), and then it will terminate successfully.

Principle of structural induction: If you always recur on smaller problems, then your procedure is sure to work.

```
  (e 3 2)
= (* 2 (e 2 2))
= (* 2 (* 2 (e 1 2)))
= (* 2 (* 2 (* 2 (e 0 2))))
= (* 2 (* 2 (* 2 1)))
= 8
```

```
(define fact
  (lambda (n)
    (if (zero? n) 1 (* n (fact (- n 1))))))
```

We can model a calculation with `fact`:

```
  (fact 4)
= (* 4 (fact 3))
= (* 4 (* 3 (fact 2)))
= (* 4 (* 3 (* 2 (fact 1))))
= (* 4 (* 3 (* 2 (* 1 (fact 0)))))
= (* 4 (* 3 (* 2 (* 1 1))))
= (* 4 (* 3 (* 2 1)))
= (* 4 (* 3 2))
= (* 4 6)
= 24
```

This is the natural recursive definition of factorial. Each call of fact is made with a promise that the value returned will be multiplied by the value of `n` at the time of the call. Thus fact is invoked in larger and larger control contexts as the calculation proceeds.

Compare this behavior with that of the following procedures.

```
(define fact-iter
  (lambda (n)
    (fact-iter-acc n 1)))

(define fact-iter-acc
  (lambda (n a)
    (if (zero? n) a (fact-iter-acc (- n 1) (* n a)))))
```

With these definitions, we calculate:

```
  (fact-iter 4)

= (fact-iter-acc 4  1)

= (fact-iter-acc 3  4)

= (fact-iter-acc 2  12)

= (fact-iter-acc 1  24)

= (fact-iter-acc 0  24)

= 24
```

Here, fact-iter-acc is always invoked in the same context (in this case, no context at all). When fact-iter-acc calls itself, it does so at the "tail end" of a call to fact-iter-acc. That is, no promise is made to do anything with the returned value other than return it as the result of the call to fact-iter-acc. This is called *tail recursion*.

Thus each step in the derivation above has the form `(fact-iter-acc` *n a*`)`.

It's not just procedure calls that cause a control stack to be necessary: it's procedure calls *inside a context* that require a control stack to keep track of the different contexts. As Guy Steele has said:

> Intuitively, [procedure] calls do not "push control stack"; instead it is argument evaluation that pushes control stack. —*Guy Steele*

This contrast will be vital later on.

Let's try this on a list processing problem.

Procedure `nth-elt`

Input: an integer *k* and a list *l*.

Output: Returns the *k*-th element of list *l*, (counting from 0). If *l* does not contain sufficiently many elements, an error is signalled.

When *k* is 0, we know the answer: `(car l)`. If *k* is greater than 0, then the answer is the $k-1$st element of `(cdr l)`. The only glitch is that we have to *guard* the `car` and `cdr` operations:

```
(define nth-elt
  (lambda (k l)
    (if (null? l)
      (error 'nth-elt "ran off end")
      (if (zero? k)
        (car l)
        (nth-elt (- k 1) (cdr l))))))
```

This is the same as the procedure `list-ref`, except that `list-ref` does not signal errors.

```
  (nth-elt 3 '(a b c d e))
= (nth-elt 2   '(b c d e))
= (nth-elt 1     '(c d e))
= (nth-elt 0       '(d e))
= d
```

Procedure: `mymap`

Input: a procedure f of one argument and a list l of values

Output: the list obtained by applying f to each element of l.

```
(define mymap
  (lambda (f l)
    (if (null? l) '()
      (cons
        (f (car l))
        (mymap f (cdr l))))))
```

This is the same as the procedure `map`.

```
(mymap mysucc '(13 26 39))
= (cons (mysucc 13) (mymap mysucc '(26 39)))
= (cons 14 (mymap mysucc '(26 39)))
= (cons 14 (cons (mysucc 26) (mymap mysucc '(39))))
= (cons 14 (cons 27 (mymap mysucc '(39))))
= (cons 14 (cons 27 (cons (mysucc 39) (mymap mysucc '()))))
= (cons 14 (cons 27 (cons 40 (mymap mysucc '()))))
= (cons 14 (cons 27 (cons 40 '())))
= '(14 27 40)
```

We will also use the multi-argument version of `map` (*q.v. infra*).

Procedure `my-memv?`

Input: a symbol *i* and a list *l* of symbols.

Output: true if *l* contains the symbol *i* as one of its members, false otherwise.

```
(define my-memv?
  (lambda (i l)
    (if (null? l) #f
        (if (eqv? i (car l)) #t
            (my-memv? i (cdr l))))))
```

Procedure `remove-first`

Input: a symbol *i* and a list *l* of symbols.

Output: A list like *l*, except that the first occurrence of *i* has been removed. If there is no occurrence of *i*, the returned list will be the same as *l*.

```
(define remove-first
  (lambda (i l)
    (if (null? l) '()
        (if (eqv? i (car l))
            (cdr l)
            (cons (car l)
                  (remove-first i (cdr l)))))))
```

Procedure `remove`

Input: a symbol *i* and a list *l* of symbols.

Output: A list the same as *l*, except that *all* occurrences of *i* have been removed. If there is no occurrence of *i*, return a list the same as *l*.

```
(define remove
  (lambda (i l)
    (if (null? l) '()
        (if (eqv? i (car l))
            (remove i (cdr l))
            (cons (car l)
                  (remove i (cdr l)))))))
```

Procedure `subst`

Input: Two symbols: `new` and `old`, and an list `slst`

Output: a list similar to `slst` but with all occurrences of `old` replaced by instances of `new`.

But wait: need to be more precise about what kind of lists are possible inputs. Let's write a grammar to describe these nested lists of symbols:

⟨s-list⟩ ::= ({⟨symbol-expression⟩}*)
⟨symbol-expression⟩ ::= ⟨symbol⟩ | ⟨s-list⟩

First we rewrite the grammar to eliminate the use of the Kleene star:

⟨s-list⟩ ::= ()
::= (⟨symbol-expression⟩ . ⟨s-list⟩)
⟨symbol-expression⟩ ::= ⟨symbol⟩ | ⟨s-list⟩

Note the "dotted-pair" notation in the second line. This notation is sometimes used as an infix version of `cons` in informal notation (not in programs). Here is means "a list whose car is a symbol-expression and whose cdr is an s-list." This notation is discussed in Dybvig's book– you should look it up there.

Now we can use the grammar to guide us in writing the program. Our follow-the-grammar pattern says we should have two procedures, one for dealing with ⟨s-list⟩ and one for dealing with ⟨symbol-expression⟩:

```
(define subst
  (lambda (new old slist)
    (if (null? slist)                    ; is the list empty?
      '()                                ; yes: then nothing to substitute
      (cons                              ; no, work on the pieces
        (subst-in-symbol-expression new old (car slist))
        (subst                       new old (cdr slist))))))

(define subst-in-symbol-expression
  (lambda (new old se)
    (if (symbol? se)
      (if (eqv? se old) new se)
      (subst new old se))))
```

Follow the Grammar!

When defining a program based on structural induction, the structure of the program should be patterned after the structure of the data.

We will see much more on this later.

Some common special forms

cond

One very common pattern of usage is the many-way branch:

```
(if test1 exp1
    (if test2 exp2
        exp_n))
```

We have a syntactic abbreviation for this pattern:

```
(cond
  (test1 exp1)                          ; eg:  ((null? l) 'foo)
  (test2 exp2)
  ...
  (else exp_n))
```

So we can rewrite `remove` as

```
(define remove
  (lambda (i l)
    (cond
      ((null? l) '())
      ((eqv? i (car l))
       (remove i (cdr l)))
      (else (cons (car l)
                  (remove i (cdr l)))))))
```

let

Lots of times we need *local names* for things. For this we use the special form `let`:

```
(let ((var1 exp1)
      (var2 exp2)
      ...)
  exp)
```

For example, we write

```
(let ((n 25))
  (let ((x (sqrt n))
        (y 3))
    (+ x y)))

= (let ((x (sqrt 25))
        (y 3))
    (+ x y))

= (let ((x 5)
        (y 3))
    (+ x y))

= (+ 5 3)

= 8
```

```
(let ((x 3))
  (let ((y (+ x 4)))
    (* x y)))

= (let ((y (+ 3 4)))
    (* 3 y))

= (let ((y 7))
    (* 3 y))

= (* 3 7) = 21
```

compare to:

```
(let ((x 3)
      (y (+ x 4)))
  (* x y))
```

Here the occurrence of `x` on the second line refers to whatever the enclosing value of `x` happens to be– maybe set by another `let`.

`Let`'s can be nested– see Sec 1.3; this is important.

Using `let`, we can give names to local procedures and pass them around:

```
(let ((f (lambda (y z) (* y (+ x z))))
      (g (lambda (u) (+ u 5))))
  (f (g 3) 17))

(let ((x 5))
  (let ((f (lambda (y z) (* y (+ x z))))
        (g (lambda (u) (+ u x)))
        (x 38))
    (f (g 3) 17)))

(let ((f (lambda (y) (y (y 5))))
      (g (lambda (z) (+ z 8))))
  (f g))

(let ((f (lambda (y z) (y (y z))))
      (g (lambda (z) (+ z 8))))
  (f g 5))
```

`map` is an extremely handy procedure. In its simplest form, it takes a procedure f of one argument and a list, and produces the list obtained by applying f to each element of l. For example

```
(map
  (lambda (x) (+ x 1))
  '(13 26 39))
=
(14 27 40)
```

`map` can also take multiple arguments. It can take a procedure f of n arguments and n lists (assumed to be of equal lengths). In this form, it produces a list whose first element is obtained by applying f to the first elements of the lists, whose second element is obtained by applying f to the second elements of the lists, etc. For example:

```
(map
  (lambda (x y z) (list x y z))
  '(11 22 33)
  '(a b c)
  '(x y z))
=
((11 a x) (22 b y) (33 c z))
```

If the lists differ in length, then the behavior is unspecified; both Chez Scheme and PLT Scheme raise an error in this case.

We will use multiple-argument `map` fairly often.