6.090, Building Programming Experience
Ben Vandiver
Lecture 4: Sugar, Recursive/Iterative, Basic Lists

Beginning of class, instructions to write a program that returns a quotient without remainder. Utilize the remainder function that we programmed before (remainder is also pre-programmed into scheme)

(quotient x y) as opposed to
Integer division x/y
(quotient 5 2) → 2

```
(define quotient
  (lambda (x y)
    (if (> x y)
        0
        (+ 1 (quotient (- x y) y)))))
```

Syntactic sugar…syntax: correct arrangement and expression of a language. Semantics of language tell you what a syntactically correct statement means.

(7 3)
Syntactically accurate and valid scheme expression. Semantically invalid.

)7 3(
Syntactically invalid.

Syntactic sugar means to "sweeten syntax", i.e., to create shortcuts

Example:

```
(define quotient
  (lambda (x y)
    (if (< x y)
        0
        (+ 1 (quotient (- x y) y)))))
```

The above is the same as the following

```
(define (quotient x y)
  (if (< x y)
      0
      (+ 1 (quotient (- x y) y))))
```

"de-sugar" if confused by new notation, and work out problem that way, too.

Special Form: Let
let introduces new variables for the duration of the body of the expression.

```
(let ((a 3))
  (+ a 2))
```

This let creates a new binding of a to 3, then evaluates the body to get 5.

```
(let ((a 3)
      (b 5))
  (+ a b)) => 8
```

Let can be understood by treating it as "syntactic sugar"

```
[(define x 7) <- binding lasts indefinitely in a define
        .
        .
        .
```
In a let, the binding is only valid for the duration of the let body.]

You may assume the existence of a lambda operation as part of the "de-sugaring" process.

```
((lambda (a b) (+a b)) 3 5)
```

The lambda operation returns a procedure. In order to fully "de-sugar" the let function, we use a combination with the newly defined lambda procedure. This procedure returns 8, the exact return of the let function.

```
(let ((a 3)
      (b (+ a 2)))
  (+ a b)) → 8
```

De-sugar

```
((lambda (a b) (+a b)) 3 (+a 2))
```

No evaluation occurs during de-sugaring.
The a in the expression (+ a 2) is not bound to anything in this case. In a let , binding is simultaneous. Values for a and b appear at same time. In lambda, bindings that occur within procedure are only valid within procedure.

```
(define a 7)
(let ((a (+ a 2))
      (b (+ a 2)))
  (+ a b)) → 18
```

The bindings in the let function are evaluated simultaneously. a = 7 travels throughout the bindings because the first binding of a does not carry over to the binding of b.

The bound values are evaluated in the expression (+ a b) simultaneously.

Special form: Let*
let* binds sequentially.

```
(define a 7)
(let* ((a (+ a 2))
       (b (+ a 2)))
  (+ a b)) → 20      ; here a gets 9, b gets 11
```

De-sugared version of the let* function:

```
(let ((a (+ a 2)))
  (let ((b (+ a 2)))
    (+ a b)))
```

The let body includes a let within.

Difference Between Recursive vs. Iterative Processes

(recursive *procedures* call themselves). Illustration of the substitution model:

```
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
```

```
(fact 3)                    not a base case
(* 3 (fact 2))              not a base case
(* 3 (* 2 (fact1)))         not a base case
(* 3 (* 2 (* 1 (fact 0))))  base case
```

Only after the base case has been reached can the multiplication of the recursive part of the definition be engaged. The innermost fact must be evaluated before any of the multiplication can begin. This causes the procedure to become very large and require a lot of memory to run. For this reason, continuously looping recursive procedures with recursive processes return the error message "maximum recursion depth exceeded".

```
(define (remainder x y)
  (if (< x y)
      x
      (remainder (- x y) y)))
```

(remainder 17 3) this is not the base case, so the recursive step is employed.
(remainder 14 3)
(remainder 11 3)
(remainder 8 3)
(remainder 5 3)
(remainder 2 3) this is the base case, so the return value is 2, and the recursion stops.

The remainder is 2.

Processes that run forever must be iterative. Deferred operations lead to crashing, because too many pending operations (as noted above) return "maximum recursion depth exceeded".

Iterative requires less space, because there are no operations to remember.

In order to identify whether a recursive procedure has a recursive or iterative process embedded in it, look to the left of the recursive call, and if there is an operation listed before the recursive function, then it is a recursive process.

Getting iterative process from recursive:

Example: Pad

Recursive (stick spaces on end after recursive call)

```
(define pad
  (lambda (s n)
    (if (= n 0)
        s
        (string-append (pad s (- n 1)) " "))))
```

Iterative (stick spaces on end before recursive call)

```
(define pad
  (lambda (s n)
    (if (= n 0)
        s
        (pad (string-append s " ") (- n 1)))))
```

Example: fact

```
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
```

Change to iterative by storing the answer-so-far in an argument to the procedure. When the base case is reached, return the answer-so-far. In the recursive case, call the procedure again with an updated answer.

```
(define (fact-helper n answer)
  (if (= n 0)
      answer
      (fact-helper (- n 1) (* n answer))))
```

```
fact 3 1                (fact-helper 0 1) base case
fact 2 3
fact 1 6
fact 0 6
```

```
(define (fact n)
  (fact-helper n 1))
```

This keeps the needed entries to fact down to one entry, which is more user-friendly than having to know what to place for the variable space labeled "answer".

```
(define (sum-to-n n)
  (if (= n 1)
      1
      (+ n (sum-to-n (- n 1)))))
```

Make this iterative and have only one necessary argument.

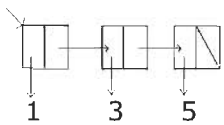
```
(define (sum-helper n answer)
  (if (= n 1)
      answer
      (sum-to-n (- n 1) (+ n answer))))

(define (sum-to-n n)
  (sum-helper n 1))
```

It takes two defines to create this result.

Data Structures

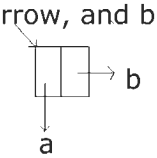
(1 3 5)



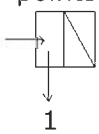
A list is a pointer to first element of list
Follow pointers to reach element in list.

Procedures

The primitive procedure cons builds a box, where a represents the value pointed to by the down arrow, and b represents the value pointed to by the right arrow.
(cons a b)

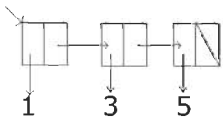


(1) pointer points to 1, then other points to end

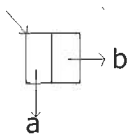


(cons 1 nil)

(cons 1 (cons 3 (cons 5 nil)))



How to, given a box, figure out what one of the pointers point to. A box is also known as a pair.



(car p) → a

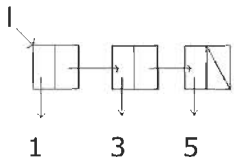
(car (cons a b)) → a

These two expressions mean exactly the same thing, where p represents some pair, which in turn is represented by (cons a b)

(cdr p) → b

(cdr (cons a b)) → b

If we take a list l pictured below, we can perform the following operations on it:



(car l) → 1
(cdr l) → (3 5)

The function list builds a list (a line of boxes with arrows as opposed to cons, which builds a single box with arrows pointing to two elements):

(list a b c) → (a b c)

first
second
third
fourth…tenth

The above commands return elements in a list when given a list as an argument. For example:

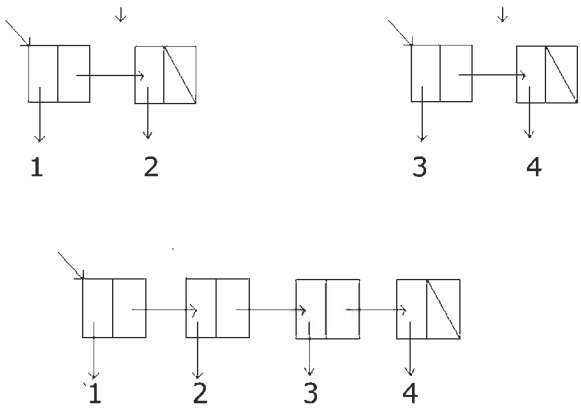
```
(define l
        (list 1 3 5))
```
(second l) → 3

list-ref returns the nth element when given a list, where the nth element is counted from left to right in a system of zero-base indexing (so in a list (1 3 5), letting n equal one would return 3, the second element in the list, and letting n equal zero would return 1, the first element in the list; note the difference from the commands first and second listed above). Here it is in scheme language:

(list-ref lst n) => nth element of list lst

(list-ref l 1) => 3

(append l1 l2) combines the lists l1 and l2
(append (list 1 2) (list 3 4)) => (1 2 3 4)



(cadr l) → (car (cdr l)) → 3

There are numerous other combinations of car and cdr: cdadr, cddadr, etc.
The output of a scheme evaluation of inputs often has the same appearance as the input, which can be confusing.

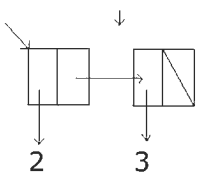
Lists are not self-evaluating. If you want a list, you must call the list procedure. Or you may call cons. The same list may be written as

(list 1) and
(cons 1 nil)

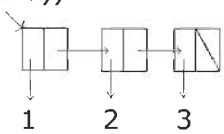
Both of the above return (1)

If you have a list and want a new one with more elements, you may stick elements onto the front of the list.

Have:
(list 2 3)

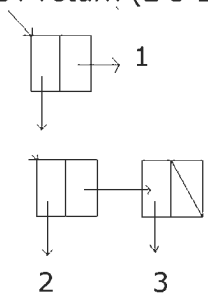


Can stick things to front of list:
(cons 1 (list 2 3))



This does not work both ways; you may not call the same procedure to add things to the back of the list, as the result will not be a straight line of boxes, but a more complicated creation.

Cannot stick things to back of list to get desired result:
(cons (list 2 3) 1) does NOT return (2 3 1); it instead returns ((2 3) . 1) which looks like:



Below is an example of a list with one element, which happens to be another list (with three elements).

((1 2 3))

It is important to be able to picture (and draw) the box-and-pointer diagram when working with lists.

New Recursive Procedure

We want to figure out the length of a list lst
Write a plan:
Base case: (list) → ( ) this is nil, the empty list. In scheme, the word nil evaluated will return the Boolean #f
So our base case is, when given a list whose value is nil, we want a return of 0 for length.
Recursive case: 1 + length (cdr lst)

As an aside, the primitive procedure null? tests to see if a list is nil, and returns a Boolean confirming its finding. Also, in scheme nil evaluates to #f, which is taken to be the empty list ( ). Thus, #f, nil, (list), and ( ) are equivalent.

Putting it all together:

```
(define (length lst)
  (if (null? lst)
      0
      (+ 1 (length (cdr lst)))))
```

Now we want a procedure that takes in a list of numbers and returns a new list with all the numbers doubled. Some example inputs and outputs:

(1 3 5) => (2 6 10)
( ) => ( )
(5) => (10)
(3 5) => (6 10)

```
(define (double-list lst)
  (if (null? lst)
      nil
      (cons (* 2 (car lst)) (double-list (cdr lst)))))
```

Do problems on Lecture 4 handout. Solutions are posted on the server under Lecture 4.