CS 61A
DISCUSSION 8

INTERPRETERS AND TAIL CALLS

Raymond Chan
Discussion 134
UC Berkeley Fall 16

AGENDA

- Announcements
- Interpreters
- Tail Calls

ANNOUNCEMENTS

- Homework 10 due tonight.
- Scheme Project due 11/17 (Start early).
  - Check website about extra credit.
- Lab 10 due Friday.
- Submit Midterm 2 Regrade Requests on Gradescope.
- Maps Composition Revision due 11/6.

MIDTERM 2

- Video Walkthrough
- https://www.youtube.com/watch?v=mfuq7fR-rpY&list=PLx38hZJ5RLZc8e46JGtPFxsDho-dbrPjO

INTERPRETERS

- Programs that understand other programs.
- Use an **underlying language** to implement an interpreter that can understand the **implemented language**.
- Read-Eval-Print-Loop (REPL).

INTERPRETERS

CALCULATOR

- In discussion today, creating interpreter for *Calculator* language.
- Subset of Scheme
- +, -, *, /
- Can be nested and take varying amounts of arguments.

INTERPRETERS

REPL

- Read
  - Lexer turns input into “tokens.”
  - Parser organizes “tokens” into data structures of the underlying language.
  - Calculator is like a Scheme List
  - To represent a Scheme List, we use Pairs, which is a form of Linked List.

INTERPRETERS

REPL

- Example: (+ 1 2)
- Can be represented as Pair('+', Pair(1, Pair(2, nil)))

INTERPRETERS

REPL

- Eval
  - Mutual recursion between eval and apply.
  - Eval: evaluates an expression according to the rules of the language.
    - Deals with **expressions**.
- Apply: applies the function to the argument values.
  - May call eval to evaluate sub-expressions.
  - Deals with **values**

INTERPRETERS

REPL

- Print displays result.

INTERPRETERS

PAIRS

- nil is an instance of the nil class. Use it as an empty list.
- Pair class
  - Similar to Link List
  - Instance attributes
    - **self.first**
    - **self.rest**

INTERPRETERS

PAIRS

- Methods
  - **__init__(self, first, second)**
  - **__len__(self)**
    - length of list
- **__getitem__(self, i)**
  - Allows *indexing* into the link list (only for Pair).
    - Do not index into a link list on the final!
- **__map__(self, fn):**
    - Applies function **fn** to all elements in the list.

INTERPRETERS

PAIRS Q1.1 - 1

Translate the following Calculator expressions into calls to the Pair constructor

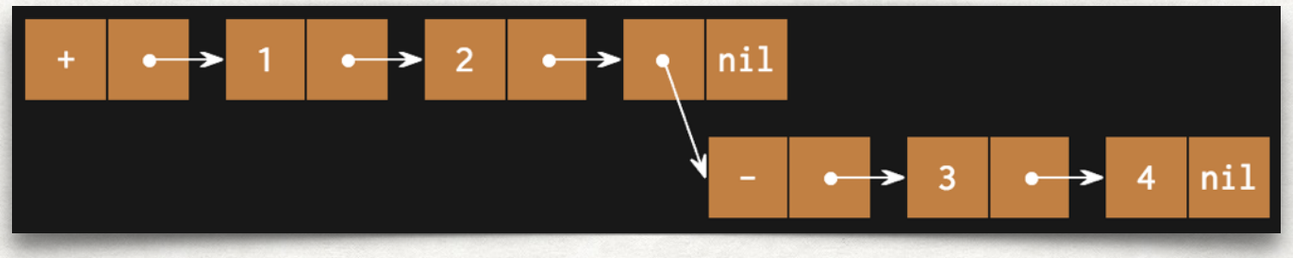
```
> (+ 1 2 (- 3 4))
```

INTERPRETERS

PAIRS Q1.1 - 1

Translate the following Calculator expressions into calls to the Pair constructor

> (+ 1 2 (- 3 4))

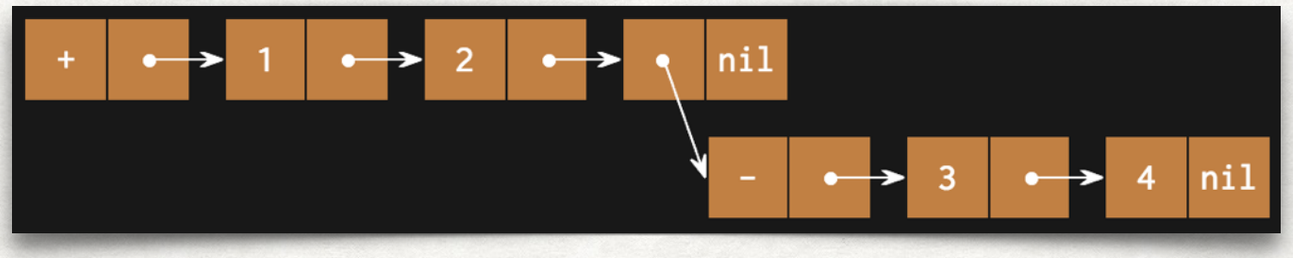


INTERPRETERS

PAIRS Q1.1 - 1

Translate the following Calculator expressions into calls to the Pair constructor

> (+ 1 2 (- 3 4))




```
Pair('+', Pair(1, Pair(2, Pair(
                                Pair('-', Pair(3, Pair(4, nil))),
                                nil))))
```

INTERPRETERS

PAIRS Q1.1 - 1

Translate the following Calculator expressions into calls to the Pair constructor

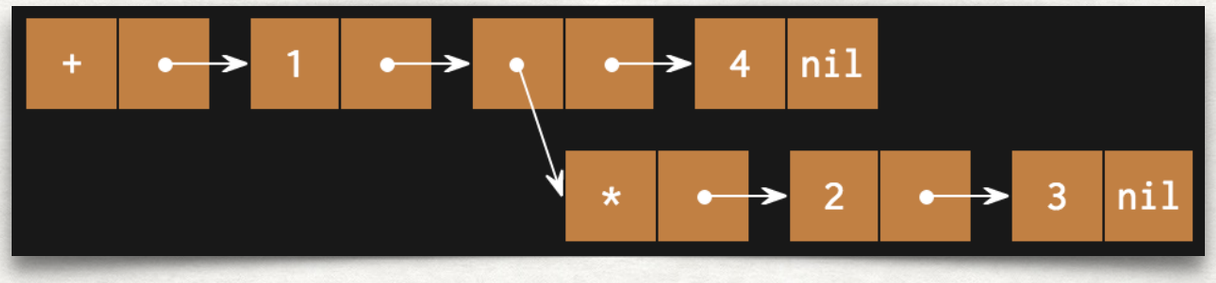
> (+ 1 (* 2 3) 4)

INTERPRETERS

PAIRS Q1.1 - 1

Translate the following Calculator expressions into calls to the Pair constructor

> (+ 1 (* 2 3) 4)

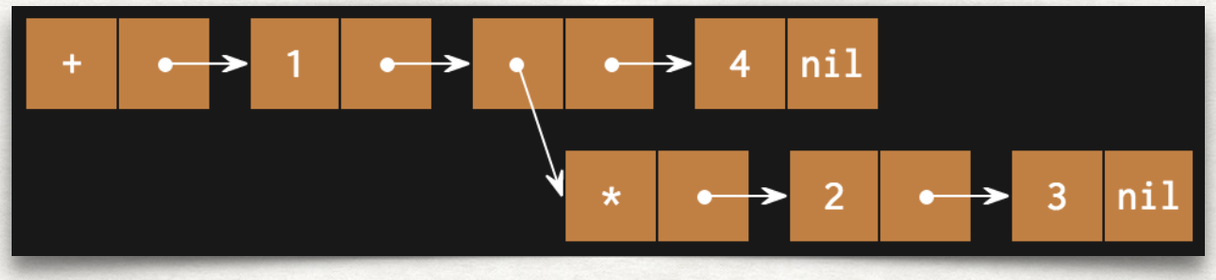


INTERPRETERS

PAIRS Q1.1 - 1

Translate the following Calculator expressions into calls to the Pair constructor

> (+ 1 (* 2 3) 4)




```
Pair('+', Pair(1, Pair(
                        Pair( '*', Pair(2, Pair(3, nil))),
                        Pair(4,nil))))
```

INTERPRETERS

PAIRS Q1.1 - 2

Translate the following Python representations of Calculator expressions into the proper Scheme syntax.

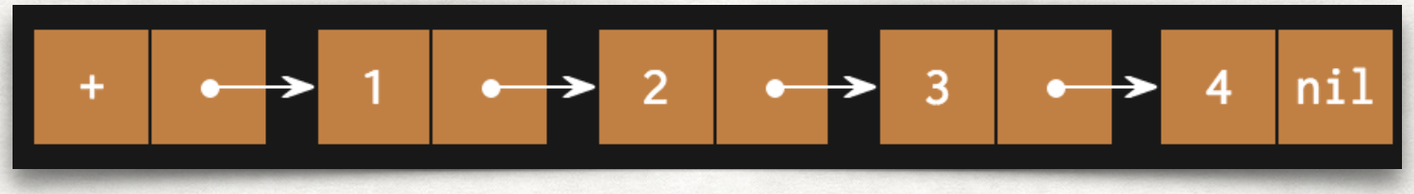
```
Pair('+', Pair(1, Pair(2, Pair(3, Pair(4, nil)))))
```

INTERPRETERS

PAIRS Q1.1 - 2

Translate the following Python representations of Calculator expressions into the proper Scheme syntax.

```
Pair('+', Pair(1, Pair(2, Pair(3, Pair(4, nil)))))
```

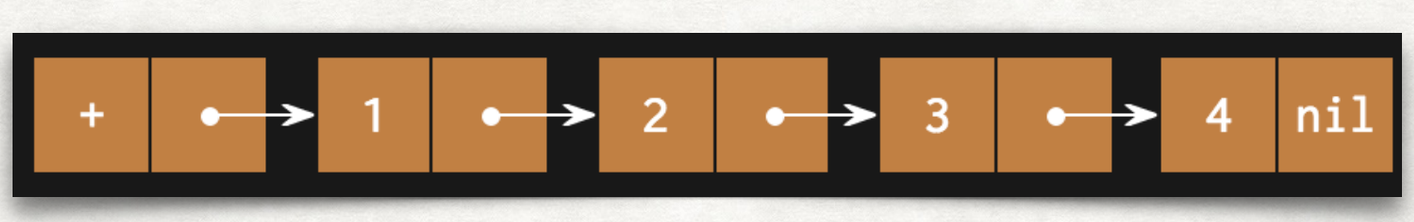


INTERPRETERS

PAIRS Q1.1 - 2

Translate the following Python representations of Calculator expressions into the proper Scheme syntax.

```
Pair('+', Pair(1, Pair(2, Pair(3, Pair(4, nil)))))
```



> (+ 1 2 3 4)

INTERPRETERS

PAIRS Q1.1 - 2

Translate the following Python representations of Calculator expressions into the proper Scheme syntax.

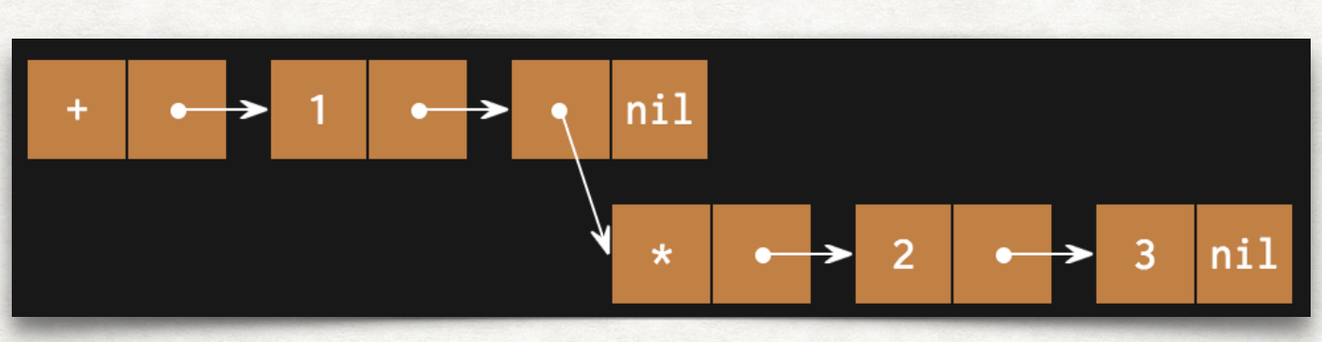
```
Pair('+', Pair(1, Pair(Pair('*', Pair(2, Pair(3, nil))), nil)))
```

INTERPRETERS

PAIRS Q1.1 - 2

Translate the following Python representations of Calculator expressions into the proper Scheme syntax.

```
Pair('+', Pair(1, Pair(Pair('*', Pair(2, Pair(3, nil))), nil)))
```

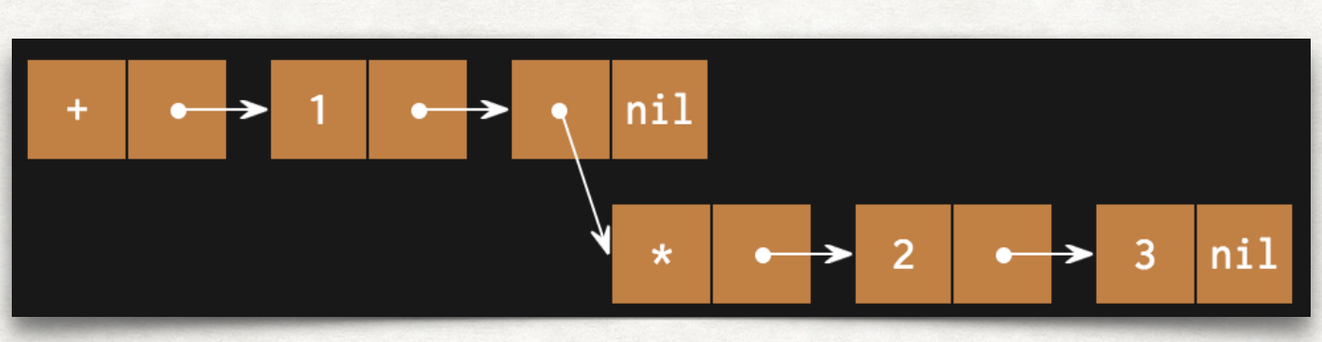


INTERPRETERS

PAIRS Q1.1 - 2

Translate the following Python representations of Calculator expressions into the proper Scheme syntax.

```
Pair('+', Pair(1, Pair(Pair('*', Pair(2, Pair(3, nil))), nil)))
```




```
> (+ 1 (* 2 3))
```

INTERPRETERS

EVALUATION

- Evaluation discovers the form of an expression and executes a corresponding evaluation rule
  - Primitive expressions evaluated directly.
  - Call expressions
    - **Evaluate** operator.
    - **Evaluate** operands from left to right.
    - **Apply** operand *values* to the operator.

TAIL CALLS

- Tail-call optimization in scheme allow recursive functions that take **constant** space.
- A **tail call** occurs if the function call is the **last operation** of the **current frame**.
- With no operations after the function call, we don't need to lookup variables anymore in the current frame.
- Can use the current frame as the function's new call frame.
- Can be a recursive call or a call to another function.

TAIL CALLS

- Factorial example; Not tail recursive

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

TAIL CALLS

- After **(fact (- n 1))** returns, we multiply the return value by **n**.
- We need to remember **n** in each frame.

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

TAIL CALLS

- Tail Recursive version.

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

TAIL CALLS

- Tail Recursive version.
- Call to fact-tail is the last operation. (- n 1) and (* n result) evaluated before apply those return values to fact-tail.

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

TAIL CALLS

- After each call to fact-tail, there are no more operations.
- We do not need the current frame’s variables anymore.

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

TAIL CALLS

- **result** is the list that we are building in each frame.
- At the end, we can just return **result**.

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

TAIL CALLS

- We keep track of **n** and **result** by passing them on as arguments to the recursive call.
- Use helper functions!

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

TAIL CALLS

- Helper function uses all required variables and additional variables as arguments.

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

TAIL CALLS

- We keep track of **n** and **result** by passing them on as arguments to the recursive call.
- The interpreter will update **n** and **result** in the current frame.

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

TAIL CALLS

- Closest thing to iteration in Scheme.

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

```
def fact(n):
    result = 1
    while n !=0:
        result = n * result
        n = n - 1
    return result
```

TAIL CALLS

IDENTIFYING TAIL CALLS

- A function call is a tail call if it is in a **tail context**. This function may or may not be tail-recursive.
- A tail-recursive function requires the recursive call to be the last action, which implies it is in a tail context.

TAIL CONTEXT

IDENTIFYING TAIL CALLS

- Last sub-expression in the body of a **lambda**.
- Second or third sub-expression in an **if** form (values that return).
- Any non-predicate sub-expression in a **cond** form.
- Last sub-expression in an **and** or an **or** form.
- Last sub-expression in a **begin**'s body.

TAIL CONTEXT

IDENTIFYING TAIL CALLS

- Last sub-expression in the body of a **lambda**.

(lambda (<parameters>) (expr_1) (expr_2) … (tail_expr))

```
> ((lambda (x) (+ 2 x) (fact-tail (- x 1))) 5)
120
```

TAIL CONTEXT

IDENTIFYING TAIL CALLS

- Second or third sub-expression in an **if** form (values that return).

```
(if <cond>
  <true_tail_expr>
  <else_tail_expr>)
```

```
(define (func x)
  (if (= 0 x)
      (fact-tail 3)
      (fact-tail 2)
  ))
```

TAIL CONTEXT

IDENTIFYING TAIL CALLS

- Any non-predicate sub-expression in a **cond** form.

```
(cond
  (<pred_expr_1> <tail_expr_1>)
  (<pred_expr_2> <tail_expr_2>)
  …
  (else <tail_expr>))
```

```
(define (func x y)
  (cond
    ((= (fact-tail 5) x) (fact-tail 3))
    ((= 20 x) (fact-tail 2))
    …
    (else (fact-tail 5))
    ))
```

TAIL CONTEXT

IDENTIFYING TAIL CALLS

- Last sub-expression in an **and** or an **or** form.

(and <expr_1> <expr_2> … <tail_expr>)
(or <expr_1> <expr_2> … <tail_expr>)

```
(define (f x y)
  (and x (eq? (fact-tail 5) y) (fact-tail 2)))

(define (g x y)
  (or #f (eq? 8 y) (fact-tail 2)))
```

TAIL CONTEXT

IDENTIFYING TAIL CALLS

- Last sub-expression in a **begin**'s body.

(begin <expr_1> <expr_2> … <tail_expr>)

(begin (+ 2 3) (- 5 3) (fact-tail 3))

TAIL CONTEXT

IDENTIFYING TAIL CALLS

- All the examples had a tail recursive function in the tail call.
- We can have non-tail recursive functions in a tail context.
  - This following expressions are not tail recursive.

i.e. (begin (+ 2 3) (fact 3) (fact 3))

i.e. (begin (+ 2 3) (fact 3) (fact-tail 3))

TAIL CONTEXT

Q1

```
(define (question-a x)
  (if (= x 0)
      0
      (+ x (question-a (- x 1)))))
```

```
(define (question-b x y)
    (if (= x 0)
        y
        (question-b (- x 1) (+ y x))))
```

```
(define (question-c x y)
    (if (> x y)
    (question-c (- y 1) x)
    (question-c (+ x 10) y)))
```

```
(define (question-d n)
      (if (question-d n)
          (question-d (- n 1))
          (question-d (+ n 10)))))
```

TAIL CONTEXT

Q1

```
(define (question-a x)
  (if (= x 0)
      0
      (+ x (question-a (- x 1)))))
```

TAIL CONTEXT

Q1

```
(define (question-a x)
  (if (= x 0)
      0
      (+ x (question-a (- x 1)))))
```

Not in tail context and therefore not tail recursive.

Need to add x with the return value of the recursive call.

Each frame remains active.

TAIL CONTEXT

Q1

```
(define (question-b x y)
    (if (= x 0)
        y
        (question-b (- x 1) (+ y x))))
```

TAIL CONTEXT

Q1

```
(define (question-b x y)
    (if (= x 0)
        y
        (question-b (- x 1) (+ y x))))
```

Both in tail context and tail recursive.
Both the if form and the
  third subexpressions are in
  tail contexts.
Last evaluated expression is recursive call.
  Thus tail recursive.
Constant frames.

TAIL CONTEXT

Q1

```
(define (question-c x y)
    (if (> x y)
    (question-c (- y 1) x)
    (question-c (+ x 10) y)))
```

TAIL CONTEXT

Q1

```
(define (question-c x y)
    (if (> x y)
    (question-c (- y 1) x)
    (question-c (+ x 10) y)))
```

In tail context and tail recursive.
The recursive calls in the
  2nd and 3rd sub-expressions
  are the last expression evaluated
  and only of them is evaluated.

TAIL CONTEXT

Q1

```
(define (question-d n)
        (if (question-d n)
            (question-d (- n 1))
            (question-d (+ n 10))))
```

TAIL CONTEXT

Q1

The recursive calls are in tail contexts and
  are both tail calls.
But conditional expression is a recursive call.
After it returns, still need to evaluate
  another recursive call.
Not **tail recursive** and does not use a constant number of frames.

```
(define (question-d n)
        (if (question-d n)
            (question-d (- n 1))
            (question-d (+ n 10))))
```

This procedure actually loops forever

RECAP

- Interpreters
- Read-Eval-Print-Loop
  - Evaluation: eval operator, operands, apply values to operator
- Tail calls allow us to use constant space in the number of frames.
- Tail calls require the last action to be a function call.

TAIL RECURSION

REVERSE - NOTES

Reverse a scheme list tail recursively.

Start off with the non-tail optimized version.

Suppose we want to reverse a linked list recursively without tail optimization, we can reverse the rest of the linked list and append the first element to the end.

e.x <1 2 3 4>. We can reverse <2 3 4> into <4 3 2> and append <1> to the end. If there is only 1 element, reversing an empty list returns nil and appending a list to the end of nil results in the list itself. (append nil '(1)) -> (1)

```
(define (reverse lst)
    (if (null> lst) lst
        (append (reverse (cdr lst))
                (list (car lst))
        )))
```

TAIL RECURSION

REVERSE - NOTES

Reverse a scheme list tail recursively.

To make it tail recursive, we introduce a so-far list that is updated at each recursive call. (Mimicking iteration)

When we update the so-far list, we want to add the current element to the front of the list. Because we are moving down the list from start to end, we are adding the beginning elements to the so-far list first and each element we see after that would come before it.

```
lst <1 2 3>
so-far ()
    lst <2 3>
    so-far (1)
        lst <3>
        so-far (2 1)
            lst <>
            so-far (3 2 1)
```

```
(define (reverse lst)
    (define (rev lst so-far)
        (if (null? lst) so-far
            (rev (cdr lst) (cons (car lst) so-far)))
    )
    (rev lst nil)
)
```

TAIL RECURSION

INSERT NOTES

Inserts element n in the correct position of a sorted Scheme list.

Starting off with the non-tail optimized version.

We want to traverse the linked list and build a new one as we move along the elements. When we finally reach the location, the new list that is built.

```
(define (insert n lst)
     (cond ((null? lst) (cons n lst))
           ((> (car lst) n) (append (list n) lst))
           (else (cons (car lst) (insert n (cdr lst))))
  )
)
```

TAIL RECURSION

INSERT NOTES

Inserts element n in the correct position of a sorted Scheme list.

Starting off with the non-tail optimized version.

Lets say we want to insert 3 into <1 2 4 5>. We need to move to the beginning of <4 5> while constructing <1 2>. Each recursive call is going to construct the current element and recursively inserting 3 into the rest of the list. When we reach the point at 4, we add 3 by appending <3> with the rest of the list (<4 5>). We don't want to make a recursive call here because you don't want to add 3 before 5 and any elements afterwards.

```
(define (insert n lst)
    (cond ((null? lst) (cons n lst))
          ((> (car lst) n) (append (list n) lst))
          (else (cons (car lst) (insert n (cdr lst))))
  )
)
```

TAIL RECURSION

INSERT NOTES

Inserts element n in the correct position of a sorted Scheme list.

Starting off with the non-tail optimized version.

Since the recursive calls are used to construct a the list before the position we want to insert, the base case will be to place n. Thus if n is greater than all elements, it will be inserted at the last place as the recursive call will traverse the whole linked list and not reach the append case.

```
(define (insert n lst)
    (cond ((null? lst) (cons n lst))
          ((> (car lst) n) (append (list n) lst))
          (else (cons (car lst) (insert n (cdr lst))))
  )
)
```

TAIL RECURSION

INSERT NOTES

Inserts element n in the correct position of a sorted Scheme list.

With the logic down we can tail optimized this with a so-far list similar to reverse.

Method 1 (using reverse)
Note: append is tail optimized.

The issue is that cons will always add the element to the beginning, or in other words, in reverse order. Thus need to reverse the return of the inner procedure.

rev-insert will recursively build rev-lst with the elements of lst in reverse order.

```
(define (insert n lst)
    (define (rev-insert lst rev-lst)
        (cond ((null? lst) (cons n rev-lst))
              ((> (car lst) n) (append (reverse lst)
                                       (cons n rev-lst)))
              (else (rev-insert (cdr lst)
                                (cons (car lst) rev-lst)))))
  (reverse (rev-insert lst nil)))
```

TAIL RECURSION

INSERT NOTES

Using the same example we want to insert 3 into <1 2 4 5>. To reach <4 5>, we would have added 1 and 2 to the starting rev-lst of nil.

lst = <4 5>
rev-lst = <2 1>

To insert 3, we would need to reverse the lst to <5 4> and then append this with 3 to rev-lst. Thats what the second case is doing.

```
(define (insert n lst)
    (define (rev-insert lst rev-lst)
        (cond ((null? lst) (cons n rev-lst))
              ((> (car lst) n) (append (reverse lst)
                                       (cons n rev-lst)))
              (else (rev-insert (cdr lst)
                                (cons (car lst) rev-lst)))))
 (reverse (rev-insert lst nil)))
```

TAIL RECURSION

INSERT NOTES

Inserts element n in the correct position of a sorted Scheme list.

With the logic down we can tail optimized this with a so-far list similar to reverse.

Method 1

Finally the base case will add n to the beginning of rev-lst when all elements are less than it.
We add to the beginning because if we don't reach the second case, we are simply reversing lst.

```
(define (insert n lst)
    (define (rev-insert lst rev-lst)
        (cond ((null? lst) (cons n rev-lst))
              ((> (car lst) n) (append (reverse lst)
                                       (cons n rev-lst)))
              (else (rev-insert (cdr lst)
                                (cons (car lst) rev-lst)))))
  (reverse (rev-insert lst nil)))
```

TAIL RECURSION

INSERT NOTES

Method 2 (without reversing)

Assumes list operator is also tail optimized (which might not be).

Depends on implementation of list with the underlying language of there interpreter.

Notice that in order the above solution to be tail optimized, append would have to be tail optimized (which it is). Knowing this we can replace the cons with append and add elements to the back of the list rather than the front.

You just have to be careful that you only call list on n or (car lst). If you call list on (cdr lst) it will be nested.

```
(define (insert n lst)
     (define (helper lst so-far)
          (cond ((null? lst) (append so-far (list n)))
                ((> (car lst) n) (append so-far (cons n lst)))
          (else (helper (cdr lst)
                        (append so-far (list (car lst)))))
   ))
  (helper lst nil))
```