Lecture #25: Scheme Examples

Translate to Scheme

- Convert this Python program into Scheme:

```
def count(predicate, L):
    if L is Link.empty:
        return 0
    elif predicate(L.first):
        return 1 + count(predicate, L.rest)
    else:
        return count(predicate, L.rest)
```

Scheme version:

Translate to Scheme

- Convert this Python program into Scheme:

```
def count(predicate, L):
    if L is Link.empty:
        return 0
    elif predicate(L.first):
        return 1 + count(predicate, L.rest)
    else:
        return count(predicate, L.rest)
```

Scheme version:

```
(define (count predicate L)
     ?

)
(count odd? '(1 12 13 19 4 6 9)) ==> 4
(count odd? '()) ==> 0
```

Translate to Scheme

- Convert this Python program into Scheme:

```
def count(predicate, L):
    if L is Link.empty:
        return 0
    elif predicate(L.first):
        return 1 + count(predicate, L.rest)
    else:
        return count(predicate, L.rest)
```

Scheme version:

```
(define (count predicate L)
    (cond ((null? L) 0)
                 ; (null? L) same as (eqv? L '()) or (eq? L '())
          ((predicate (car L))
           (+ 1 (count predicate (cdr L))))
          (else (count predicate (cdr L))))  ; in cond, else == #t
)
(count odd? '(1 12 13 19 4 6 9)) ==> 4
(count odd? '()) ==> 0
```

- Is this tail-recursive?

Translate to Scheme

- Convert this Python program into Scheme:

```python
def count(predicate, L):
    if L is Link.empty:
        return 0
    elif predicate(L.first):
        return 1 + count(predicate, L.rest)
    else:
        return count(predicate, L.rest)
```

Scheme version:

```scheme
(define (count predicate L)
    (cond ((null? L) 0)
          ((predicate (car L))
           (+ 1 (count predicate (cdr L))))  ; Not a tail call
          (else (count predicate (cdr L))))  ; in cond, else == #t
)
(count odd? '(1 12 13 19 4 6 9)) ==> 4
(count odd? '()) ==> 0
```

- Is this tail-recursive? No

Review of Iteration via Tail Recursion

- Earlier in the course, we saw that iterations are related to tail-recursions.
- Consider a general Python loop:

```
def my_function(...):
    <variables> = <initial values>
    while <some condition>:
        <variables> = <new values>
    return <some value>
```

- Many programs can be put into this form, equivalent to

```
def my_function(...):
    def looper(<variables>):
        if <some condition>:
            return looper(<new values>)
        else:
            return <some value>
    return looper(<initial values>)
```

Review of Iteration via Tail Recursion (II)

- And this Python recursion:

```
def my_function(...):
    def looper(<variables>):
        if <some condition>:
            return looper(<new values>)
        else:
            return <some value>
    return looper(<initial values>)
```

- Converts directly into Scheme:

```
(define (my_function ...)
    (define (looper <variables>)
       (if <some condition> (looper <new values>)
           <some value>))
    (looper <initial values>))
```

- Significance of this particular kind of recursion is that Scheme implementations (but not Python) must not fail regardless of the depth of the tail calls.

Tail-Recursive Version of count

- First, the Python version:

```
def count(predicate, L):
    ?
```

Tail-Recursive Version of count

- First, the Python version:

```
def count(predicate, L):
    def count1(L, s):
        """Return S + # of items in L that satisfy PREDICATE."""
        ?
    return count1(L, 0)
```

Tail-Recursive Version of count

- First, the Python version:

```
def count(predicate, L):
    def count1(L, s):
        """Return S + # of items in L that satisfy PREDICATE."""
        if L is Link.empty:
            return s
        elif predicate(L.first):
            return count1(L.rest, s + 1)
        else:
            return count1(L.rest, s)
    return count1(L, 0)
```

Tail-Recursive Version of count

- First, the Python version:

```
def count(predicate, L):
    def count1(L, s):
        """Return S + # of items in L that satisfy PREDICATE."""
        if L is Link.empty:
            return s
        elif predicate(L.first):
            return count1(L.rest, s + 1)
        else:
            return count1(L.rest, s)
    return count1(L, 0)
```

- And now, Scheme:

```
(define (count predicate L)
    ?
)
```

Tail-Recursive Version of count

- First, the Python version:

```
def count(predicate, L):
    def count1(L, s):
        """Return S + # of items in L that satisfy PREDICATE."""
        if L is Link.empty:
            return s
        elif predicate(L.first):
            return count1(L.rest, s + 1)
        else:
            return count1(L.rest, s)
    return count1(L, 0)
```

- And now, Scheme:

```
(define (count predicate L)
    (define (count1 L s)
        ?)
    (count1 L 0)
)
```

Tail-Recursive Version of count

- First, the Python version:

```
def count(predicate, L):
    def count1(L, s):
        """Return S + # of items in L that satisfy PREDICATE."""
        if L is Link.empty:
            return s
        elif predicate(L.first):
            return count1(L.rest, s + 1)
        else:
            return count1(L.rest, s)
    return count1(L, 0)
```

- And now, Scheme:

```
(define (count predicate L)
    (define (count1 L s)
        (cond ((null? L) s)
              ((predicate (car L)) (count1 (cdr L) (+ s 1)))
              (#t (count1 (cdr L) s))))
    (count1 L 0)
)
```

Another Higher-Order Function Example: Map

- We've seen `map` in Python, where it is built-in for iterables, and we can define it there for linked lists:

```
def map(fn, L):
    if L is Link.empty:
        return Link.empty
    else:
        return Link(fn(L.first), map(fn, L.rest))
```

- What about in Scheme?

Another Higher-Order Function Example: Map

- We've seen `map` in Python, where it is built-in for iterables, and we can define it there for linked lists:

```
def map(fn, L):
    if L is Link.empty:
        return Link.empty
    else:
        return Link(fn(L.first), map(fn, L.rest))
```

- What about in Scheme?

```
scm> (define (map fn L)
)
```

Another Higher-Order Function Example: Map

- We've seen `map` in Python, where it is built-in for iterables, and we can define it there for linked lists:

```
def map(fn, L):
    if L is Link.empty:
        return Link.empty
    else:
        return Link(fn(L.first), map(fn, L.rest))
```

- What about in Scheme?

```
scm> (define (map fn L)
        (if (null? L)
        )
     )
```

Another Higher-Order Function Example: Map

- We've seen `map` in Python, where it is built-in for iterables, and we can define it there for linked lists:

```
def map(fn, L):
    if L is Link.empty:
        return Link.empty
    else:
        return Link(fn(L.first), map(fn, L.rest))
```

- What about in Scheme?

```
scm> (define (map fn L)
    (if (null? L)
        (cons (fn (car L)) (map fn (cdr L)))
    )
)
```

Another Higher-Order Function Example: Map

- We've seen `map` in Python, where it is built-in for iterables, and we can define it there for linked lists:

```
def map(fn, L):
    if L is Link.empty:
        return Link.empty
    else:
        return Link(fn(L.first), map(fn, L.rest))
```

- What about in Scheme?

```
scm> (define (map fn L)
        (if (null? L)
            (cons (fn (car L)) (map fn (cdr L)))
        )
     )
scm> (map - '(1 2 3))
(-1 -2 -3)
```

Tail-Recursive Map?

- Map is a little tricky to make tail-recursive.
- Obvious way would be to pass the initial part of the translated list as a parameter in an inner recursive procedure:

```
(define (map fn L)
    (define (loop list-so-far L)
        (if (null? L) list-so-far
            ???)) ; What goes wrong here?
    (loop '() L))
```

- Mutation of the last pair in the list would come in handy here, but we're trying to avoid that.
- So how about

```
(define (map fn L)
    (define (loop list-so-far L)
        (if (null? L) list-so-far
            (loop (append list-so-far (list (fn (car L)))) (cdr L))))
    (loop '() L))
```

where `append` is like Python's `.extend`, but for linked lists.

- Why is this horrendous?

Reverse

- Suppose we could write `(reverse L)` to get the reverse of a list:

```
scm> (reverse '(1 2 3))
(3 2 1)
```

- How could we use this to do map tail-recursively?

```
(define (map fn L)
    (define (loop list-so-far L)
        (if (null? L) list-so-far
            ?))
    ?)
```

- So now we just have to get a tail-recursive `reverse`

Reverse

- Suppose we could write `(reverse L)` to get the reverse of a list:

```
scm> (reverse '(1 2 3))
(3 2 1)
```

- How could we use this to do map tail-recursively?

```
(define (map fn L)
    (define (loop list-so-far L)
        (if (null? L) list-so-far
            (loop (cons (fn (car L)) list-so-far) (cdr L))))
    ?)
```

- So now we just have to get a tail-recursive `reverse`

Reverse

- Suppose we could write `(reverse L)` to get the reverse of a list:

```
scm> (reverse '(1 2 3))
(3 2 1)
```

- How could we use this to do map tail-recursively?

```
(define (map fn L)
    (define (loop list-so-far L)
        (if (null? L) list-so-far
            (loop (cons (fn (car L)) list-so-far) (cdr L))))
    (reverse (loop '() L)))
```

- So now we just have to get a tail-recursive `reverse`

Tail-Recursive Reverse

- Not really so difficult, once you think about how you realize that, for example,

```
scm> (define L '(1 2 3))
scm> (reverse L)
(3 2 1)
scm> (cons (car (cdr (cdr L))) (cons (car (cdr L)) (cons (car L) '())))
(3 2 1)
```

- This might suggest the order in which the reversed list gets built, suggesting a program like this:

```
(define (reverse L)
    (define (reverse1 ?)
        ?)
    ?)
```

Tail-Recursive Reverse

- Not really so difficult, once you think about how you realize that, for example,

```
scm> (define L '(1 2 3))
scm> (reverse L)
(3 2 1)
scm> (cons (car (cdr (cdr L))) (cons (car (cdr L)) (cons (car L) '())))
(3 2 1)
```

- This might suggest the order in which the reversed list gets built, suggesting a program like this:

```
(define (reverse L)
    (define (reverse1 so-far L)
        (if (null? L) so-far
            (reverse1 (cons (car L) so-far) (cdr L))))
    ?)
```

Tail-Recursive Reverse

- Not really so difficult, once you think about how you realize that, for example,

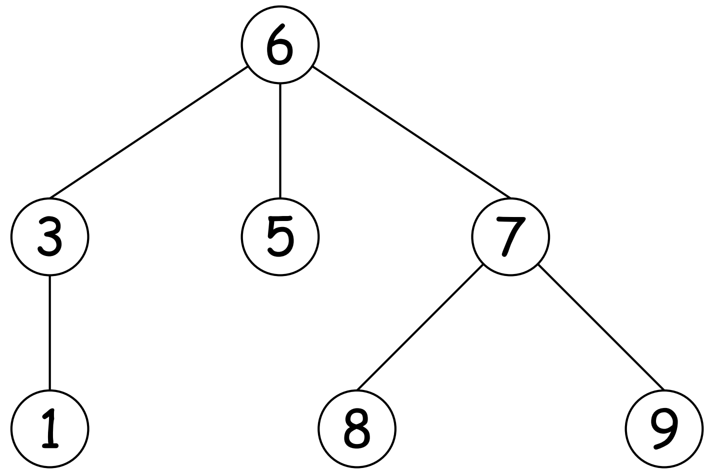
```
scm> (define L '(1 2 3))
scm> (reverse L)
(3 2 1)
scm> (cons (car (cdr (cdr L))) (cons (car (cdr L)) (cons (car L) '())))
(3 2 1)
```

- This might suggest the order in which the reversed list gets built, suggesting a program like this:

```
(define (reverse L)
    (define (reverse1 so-far L)
        (if (null? L) so-far
            (reverse1 (cons (car L) so-far) (cdr L))))
    (reverse1 '() L))
```

Trees

- How could we represent a tree in Scheme?



- Can use a representation similar to what we used in Python, such as

```
(6 (3 (1)) (5) (7 (8) (9)))
```

- Abstracting into functions:

```
(define (tree label children) (cons label children))
(define (label tr) (car tr))
(define (children tr) (cdr tr))
(define (is-leaf tr) (null? (cdr tr)))
```

Tree Recursions

- Assuming our labels are integers, how could we implement the label-doubling function from lecture 12 in Scheme?

```
(define (double tr)
     "Return a tree identical to TR, but with all labels doubled."
     ?
)

(define aTree (tree 6
                    (list (tree 3 (list (tree 1 '())))
                          (tree 5 '())
                          (tree 7 (list (tree 8 '()) (tree 9 '()))))))

aTree ==> (6 (3 (1)) (5) (7 (8) (9)))
(double aTree) ==> (12 (6 (2)) (10) (14 (16) (18)))
```

Tree Recursions

- Assuming our labels are integers, how could we implement the label-doubling function from lecture 12 in Scheme?

```
(define (double tr)
     "Return a tree identical to TR, but with all labels doubled."
     (tree (* (label tr) 2) (map double (children tr)))
)

(define aTree (tree 6
                    (list (tree 3 (list (tree 1 '())))
                          (tree 5 '())
                          (tree 7 (list (tree 8 '()) (tree 9 '()))))))

aTree ==> (6 (3 (1)) (5) (7 (8) (9)))
(double aTree) ==> (12 (6 (2)) (10) (14 (16) (18)))
```