Writing Functions in Scheme

- Suppose we want a function **ct** which takes a list of symbols and returns the number of symbols in the list

(**ct '**(**a b c**)) $\longrightarrow$ 3

(**ct '**()) $\longrightarrow$ 0

(**ct '**(**x y z w t**)) $\longrightarrow$ 5

How can we write this function?

Writing Functions in Scheme

- Answer #1: Have the instructor write it

```
;; ct : <list-of-sym>   ->   <num>
;;  (ct '()) →→ 0
;;  (ct '(a b c)) →→ 3
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))
```

Checking My Answer: Empty List

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(ct '())
```

$\rightarrow$

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

  (cond
    [(null? '()) 0]
    [else (+ 1 (ct (cdr '())))])
```

Checking My Answer: Empty List

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

  (cond
    [(null? '()) 0]
    [else (+ 1 (ct (cdr '())))])
```

→

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

  (cond
    [#t 0]
    [else (+ 1 (ct (cdr '())))])
```

Checking My Answer: Empty List

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

  (cond
    [#t 0]
    [else (+ 1 (ct (cdr '())))])
```

→

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

0
```

Checking My Answer: List of 3 Symbols

```
(define (ct l)                    →   (define (ct l)
  (cond                                  (cond
    [(null? l) 0]                          [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))            [else (+ 1 (ct (cdr l)))]))

(ct '(a b c))                            (cond
                                           [(null? '(a b c)) 0]
                                           [else (+ 1 (ct (cdr '(a b c))))])
```

Checking My Answer: List of 3 Symbols

```
(define (ct l)                                 →    (define (ct l)
  (cond                                               (cond
    [(null? l) 0]                                       [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))                         [else (+ 1 (ct (cdr l)))]))

  (cond                                               (cond
    [(null? '(a b c)) 0]                                [#f 0]
    [else (+ 1 (ct (cdr '(a b c))))])                   [else (+ 1 (ct (cdr '(a b c))))])
```

Checking My Answer: List of 3 Symbols

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

  (cond
    [#f 0]
    [else (+ 1 (ct (cdr '(a b c))))])
```

→

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(+ 1 (ct (cdr '(a b c))))
```

Checking My Answer: List of 3 Symbols

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(+ 1 (ct (cdr '(a b c))))
```

$\rightarrow$

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(+ 1
   (ct '(b c)))
```

Checking My Answer: List of 3 Symbols

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(+ 1
   (ct '(b c)))
```

→

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(+ 1
     (cond
       [(null? '(b c)) 0]
       [else (+ 1 (ct (cdr '(b c))))]))
```

Checking My Answer: List of 3 Symbols

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(+ 1
    (cond
      [(null? '(b c)) 0]
      [else (+ 1 (ct (cdr '(b c))))]))
```

→

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(+ 1
    (cond
      [#f 0]
      [else (+ 1 (ct (cdr '(b c))))]))
```

Checking My Answer: List of 3 Symbols

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(+ 1
   (cond
     [#f 0]
     [else (+ 1 (ct (cdr '(b c))))]))
```

→

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(+ 1
   (+ 1
      (ct (cdr '(b c)))))
```

Checking My Answer: List of 3 Symbols

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(+ 1
   (+ 1
      (ct (cdr '(b c)))))
```

→

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(+ 1
   (+ 1
      (ct '(c))))
```

Checking My Answer: List of 3 Symbols

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(+ 1
   (+ 1
      (ct '(c))))
```

→

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(+ 1
   (+ 1
        (cond
          [(null? '(c)) 0]
          [else (+ 1 (ct (cdr '(c))))])))
```

Checking My Answer: List of 3 Symbols

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(+ 1
   (+ 1
       (cond
         [(null? '(c)) 0]
         [else (+ 1 (ct (cdr '(c))))])))
```

→

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(+ 1
   (+ 1
       (cond
         [#f 0]
         [else (+ 1 (ct (cdr '(c))))])))
```

Checking My Answer: List of 3 Symbols

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(+ 1
   (+ 1
        (cond
          [#f 0]
          [else (+ 1 (ct (cdr '(c))))])))
```

→

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(+ 1
   (+ 1
      (+ 1
         (ct (cdr '(c))))))
```

Checking My Answer: List of 3 Symbols

```
(define (ct l)                        →  (define (ct l)
  (cond                                    (cond
    [(null? l) 0]                            [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))              [else (+ 1 (ct (cdr l)))]))

(+ 1                                     (+ 1
   (+ 1                                     (+ 1
      (+ 1                                     (+ 1
         (ct (cdr '(c))))))                       (ct '()))))
```

Checking My Answer: List of 3 Symbols

```
(define (ct l)                         →   (define (ct l)
  (cond                                      (cond
    [(null? l) 0]                              [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))                [else (+ 1 (ct (cdr l)))]))

(+ 1                                       (+ 1
   (+ 1                                       (+ 1
      (+ 1                                       (+ 1
         (ct '()))))                                  (cond
                                                        [(null? '()) 0]
                                                        [else (+ 1 (ct (cdr '())))]))))
```

Checking My Answer: List of 3 Symbols

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(+ 1
   (+ 1
      (+ 1
           (cond
             [(null? '()) 0]
             [else (+ 1 (ct (cdr '())))]))))
```

→

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(+ 1
   (+ 1
      (+ 1
           (cond
             [#t 0]
             [else (+ 1 (ct (cdr '())))]))))
```

Checking My Answer: List of 3 Symbols

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(+ 1
   (+ 1
      (+ 1
         (cond
           [#t 0]
           [else (+ 1 (ct (cdr '())))]))))
```

→

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(+ 1
   (+ 1
      (+ 1
         0)))
```

Checking My Answer: List of 3 Symbols

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(+ 1
   (+ 1
      (+ 1
         0)))
```

→

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(+ 1
   (+ 1
      1))
```

Checking My Answer: List of 3 Symbols

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(+ 1
   (+ 1
      1))
```

→

```
(define (ct l)
  (cond
    [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))

(+ 1
   2)
```

Checking My Answer: List of 3 Symbols

```
(define (ct l)                    →    (define (ct l)
  (cond                                  (cond
    [(null? l) 0]                          [(null? l) 0]
    [else (+ 1 (ct (cdr l)))]))            [else (+ 1 (ct (cdr l)))]))

(+ 1                                   3
   2)
```

Writing Functions in Scheme: Answer #2

Answer #2: Use the general design recipe

- Locate or write a data definition
- Write a contract
- Write examples
- Create a template that follows the shape of the data definition
- Convert the template to the final function
- Run examples as tests

Writing Functions in Scheme: Answer #2

Answer #2: Use the general design recipe

- Locate or write a data definition
- Write a contract
- Write examples
- Create a template that follows the shape of the data definition
- Convert the template to the final function
- Run examples as tests

works 90% of the time

Data Definitions

What is a "list of symbols"?

<list-of-sym> ::= **'**()
::= (**cons** <symbol> <list-of-sym>)

- Sometimes the ***data definition*** is given, somtimes you have to create it
- Usually include it in your code as a comment

Contracts

A ***contract*** is a comment that identifies set of input values and output values

;; **ct**: <list-of-sym> **->** <num>

- All mentioned data sets should have a data definition somewhere

Examples

Examples (usually in comments at first) help clarify the purpose of the function

;; (**ct '**()) $\longrightarrow$ 0
;; (**ct '**(**a b c**)) $\longrightarrow$ 3

- Make sure that every case in the data definition is covered at least once

Template

A ***template*** reflects the structure of the input according to the data definition

<list-of-sym> ::= **'**()
::= (**cons** <symbol> <list-of-sym>)

```
(define (ct l)
  (cond
   [(null? l) ...]
   [(pair? l) ...(car l)...(ct (cdr l))...]))
```

Template

A ***template*** reflects the structure of the input according to the data definition

```
<list-of-sym>  ::=  '()
               ::=  (cons <symbol> <list-of-sym>)
```

```
(define (ct l)
  (cond
   [(null? l) ...]
   [(pair? l) ...(car l)...(ct (cdr l))...]))
```

- Two cases in data definition implies **cond** with two cond-lines

Template

A ***template*** reflects the structure of the input according to the data definition

<list-of-sym> ::= '()
::= (**cons** <symbol> <list-of-sym>)

```
(define (ct l)
  (cond
   [(null? l) ...]
   [(pair? l) ...(car l)...(ct (cdr l))...]))
```

- Corresponding predicate for each data case

Template

A ***template*** reflects the structure of the input according to the data definition

<list-of-sym> ::= '()
::= (**cons** <symbol> <list-of-sym>)

```
(define (ct l)
  (cond
   [(null? l) ...]
   [(pair? l) ...(car l)...(ct (cdr l))...]))
```

- Extract parts in cases with meta-variables

Template

A ***template*** reflects the structure of the input according to the data definition

<list-of-sym> ::= '()
::= (**cons** <symbol> <list-of-sym>)

```
(define (ct l)
  (cond
   [(null? l) ...]
   [(pair? l) ...(car l)...(ct (cdr l))...]))
```

- Recursive call for self-references in data definition

Template

A ***template*** reflects the structure of the input according to the data definition

<list-of-sym> ::= '()
::= (**cons** <symbol> <list-of-sym>)

```
(define (ct l)
  (cond
   [(null? l) ...]
   [(pair? l) ...(car l)...(ct (cdr l))...]))
```

- A template depends only on the input data; it ignores the function's purpose

(Nevertheless, generating a template, which is fairly automatic, usually provides most of the function)

Template to Function

Transform template to function line-by-line

```
(define (ct l)
  (cond
   [(null? l) ...]
   [(pair? l) ...(car l)...(ct (cdr l))...]))
```

Template to Function

Transform template to function line-by-line

```
(define (ct l)
  (cond
   [(null? l) 0]
   [(pair? l) ...(car l)...(ct (cdr l))...]))
```

Template to Function

Transform template to function line-by-line

```
(define (ct l)
  (cond
   [(null? l) 0]
   [(pair? l) (+ 1 (ct (cdr l)) )]))
```

- Sometimes, a part of the template isn't needed

Reminder: Recipe

- Locate or write a data definition
- Write a contract
- Write examples
- Create a template that follows the shape of the data definition
- Convert the template to the final function
- Run examples as tests

Reminder: Template Steps

- Create a **cond** expression with one line for each case in the data definition
- Write down a predicate for each case
- For the answer, extract parts in cases with meta-variables
- For each self-reference in the data definition, add a recursive call

Shape of template shape == Shape of data definition

More Examples

(more examples in class)

Generalized Recipe

- Locate or write data definitions
- Write contracts
- Write examples
- Create a template that follows the shape of the data definition, one for each data definition
- Convert the templates to the final functions
- Run examples as tests