Pairs, Lists, Trees

Scheme supports **Pairs**

$$\texttt{cons} : T_1, T_2 \to \underbrace{T_1 \times T_2}_{\text{types of pairs}}$$

$$\begin{aligned}\texttt{car} &: T_1 \times T_2 \to T_1\\ \texttt{cdr} &: T_1 \times T_2 \to T_2\end{aligned}$$

Pairing Contract

$$\begin{aligned}\texttt{(car (cons } \langle \mathrm{v}_1 \rangle\ \langle \mathrm{v}_2 \rangle\texttt{))} &= \langle \mathrm{v}_1 \rangle\\ \texttt{(cdr (cons } \langle \mathrm{v}_1 \rangle\ \langle \mathrm{v}_2 \rangle\texttt{))} &= \langle \mathrm{v}_2 \rangle\end{aligned}$$

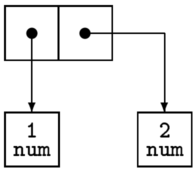
$$\texttt{(pair? } \langle \mathrm{p} \rangle\texttt{)} \Longrightarrow \begin{cases} \texttt{\#t} & \text{if } \langle \mathrm{p} \rangle \text{ is a pair (a \textit{cons-cell});}\\ \texttt{\#f} & \text{otherwise} \end{cases}$$

Three Descriptions of Pairs

Input Expression

```
(cons 1 2)
```

Box and Pointer



Output Representation

$$\underbrace{\texttt{(1 . 2)}}_{\text{dotted pair notation}}$$

Definition:

1. The "empty list", `nil`, is a list of type T, for any T.
2. If $\langle \mathrm{L} \rangle$ is a list (of elements of) type T, and $\langle \mathrm{v} \rangle$ is a value of type T, then

   $$\texttt{(cons } \langle \mathrm{v} \rangle \ \langle \mathrm{L} \rangle\texttt{)}$$

   is a list (of elements of) type T.

Can be expressed with a Type Equation:

$$\mathrm{List}(T) = \{\texttt{nil}\} \uplus (T \times \mathrm{List}(T))$$

The List Contract

$$\texttt{(null? nil)} \Longrightarrow \texttt{\#t}$$
$$\texttt{(null? } \langle\mathrm{v}\rangle\texttt{)} \Longrightarrow \texttt{\#f}, \quad \text{if } \langle\mathrm{v}\rangle \neq \texttt{nil}$$

$$\texttt{(car (cons } \langle\mathrm{v}\rangle\ \langle\mathrm{L}\rangle\texttt{))} \Longrightarrow \langle\mathrm{v}\rangle$$
$$\texttt{(cdr (cons } \langle\mathrm{v}\rangle\ \langle\mathrm{L}\rangle\texttt{))} \Longrightarrow \langle\mathrm{L}\rangle$$

Same contract as pairing:
In Scheme, non-null Lists are a Special case of Pairs.

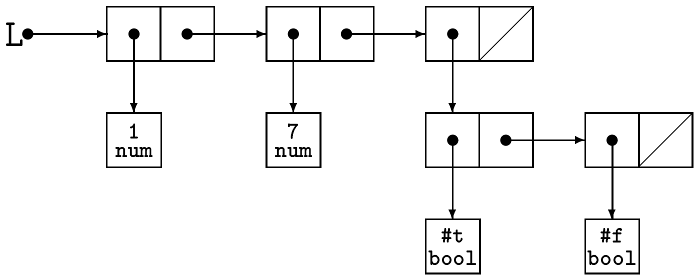
Scheme supports lists

(list $\langle \mathrm{v}_0 \rangle$ $\cdots$ $\langle \mathrm{v}_\mathrm{n} \rangle$) =
    (cons $\langle \mathrm{v}_0 \rangle$ (cons ... (cons $\langle \mathrm{v}_\mathrm{n} \rangle$ nil) ...)

Input Expression:

```
(define L (list 1 (+ 3 4) (list #t #f)))
```

Box & Pointer:



Output Representation:

```
(1 7 (#t #f))
```

Scheme supports lists

`'()` $\Longrightarrow$ the empty list

`(length (list 1 2 4))` $\Longrightarrow$ `3`

`(append (list 3 4) (list 6 8))`
$\Longrightarrow \underbrace{\texttt{(3 4 6 8)}}_{\texttt{output representation}}$

`(list-ref (list` $\langle \mathrm{v}_0 \rangle$ $\langle \mathrm{v}_1 \rangle$ $\langle \mathrm{v}_2 \rangle$ $\cdots$ `) k)` $\Longrightarrow$ $\langle \mathrm{v}_\mathrm{k} \rangle$

`(list? (list 3 (lambda (x) x)))` $\Longrightarrow$ `#t`

`(list? (cons 1 2))` $\Longrightarrow$ `#f`

These list ops are “built-in” but could all be defined using the list primitives:

`pair?  null?  car  cdr  cons  nil`

For example,

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

(define (list-ref l n)
  (if (zero? n)
      (car l)
      (list-ref (cdr l) (- n 1))))

(define (list? l)
  (or (null? l)
      (and (pair? l) (list? (cdr l)))))
```

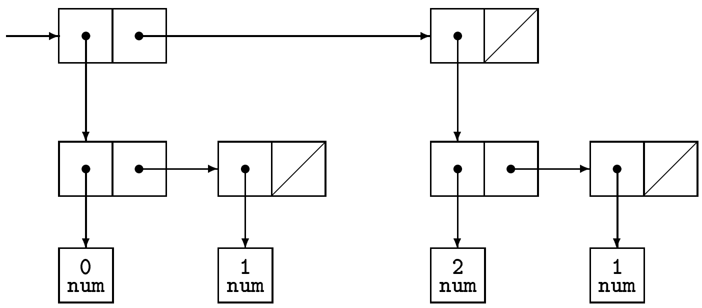
`length`, `list?` take $O(n)$ steps on a list of length n.

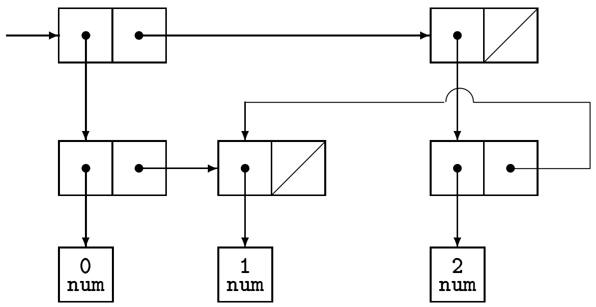
`(list-ref l k)` takes $O(k)$ steps.

Box and Pointer gives more information than the output representation of a list

Input	**Output**
`(list (list 0 1)` `      (list 2 1))`	`((0 1) (2 1))`

`(let ((l1 (list 1)))`
`  (list (cons 0 l1)`
`        (cons 2 l1)))`
	`((0 1) (2 1))`


0
num
1
num
2
num
1
num


0
num
1
num
2
num

List Abstractions

$$\texttt{(map } \langle\text{proc}\rangle \texttt{ (list } \langle\text{v}_0\rangle \; \langle\text{v}_1\rangle \; \cdots \; \langle\text{v}_\text{n}\rangle\texttt{))}$$
$$\Longrightarrow$$
$$\begin{array}{l} \texttt{(list (}\langle\text{proc}\rangle \; \langle\text{v}_0\rangle\texttt{)} \\ \qquad\;\; \texttt{(}\langle\text{proc}\rangle \; \langle\text{v}_1\rangle\texttt{)} \\ \qquad\qquad\quad \vdots \\ \qquad\;\; \texttt{(}\langle\text{proc}\rangle \; \langle\text{v}_\text{n}\rangle\texttt{))} \end{array}$$

```
(map square (list 1 2 3 4))
    ⟹ (1 4 9 16)
```

```
(map list (list 1 2 3 4))
    ⟹ ((1) (2) (3) (4))
```

Filter out "good" elements

```
(filter odd? (list 1 2 3 4 5))
    ⟹ (1 3 5)
```

```
(filter even? (list 1 2 3 4 5))
    ⟹ (2 4)
```

```
(define (filter pred lst)
  (cond ((null? lst) nil)
        ((pred (car lst))
         (cons (car lst)
               (filter pred (cdr lst))))
        (else (filter pred (cdr lst)))))
```

Enumerate Integers

```
(define (integers-between low high)
  (if (> low high)
      nil
      (cons low (integers-between (+ low 1)
                                  high))))

(integers-between 5 9) -> (5 6 7 8 9)
```

Examples

```
(map fib (integers-between 10 20))) -> (55 89 ... 6765)
```

```
(filter even? (map fib (integers-between 10 20)))
```

```
(map fib (filter even?  (integers-between 10 20)))
```

```
(accumulate ⟨op⟩ ⟨init⟩ (list ⟨v0⟩ ⟨v1⟩ ··· ⟨vn⟩))
    ⟹
    (⟨op⟩ ⟨v0⟩ (⟨op⟩ ⟨v1⟩ (⟨op⟩ ··· (⟨op⟩ ⟨vn⟩ ⟨init⟩))))

(accumulate + 0 (list 1 2 5 6))
    ⟹ (+ 1 (+ 2 (+ 5 (+ 6 0))))
    ⟹ 14
```

```
(accumulate op init lst)
```

$$
\begin{aligned}
\texttt{lst} &: \mathrm{List}(T) \\
\texttt{init} &: S \\
\texttt{op} &: (T, S) \to S
\end{aligned}
$$

$$
\texttt{accumulate} : \big((T, S) \to S,\ S,\ \mathrm{List}(T)\big) \to S
$$

Accumulation

```
(define (accumulate op init lst)
  (if (null? lst)
      init
      (op (car lst)
          (accumulate op init (cdr lst)))))
```

Sum as Accumulation

```
(define (addem-up lst)
  (accumulate + 0 lst))
```

Length as Accumulation

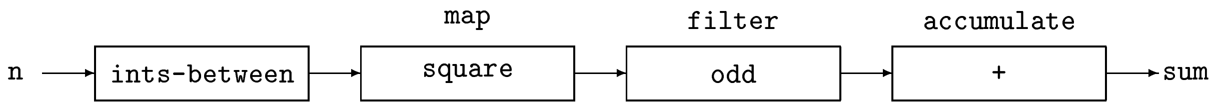
```
(define (length lst)
  (accumulate (lambda (x y) (+ 1 y))
              0
              lst))
```

Append as Accumulation

```
(define (append list1 list2)
  (accumulate cons list2 list1))
```

Using the List Abstractions

```
(define (sum-of-odd-squares n)
  (accumulate + 0                                        ; sum
            (filter odd?                                 ; odd
                    (map square                          ; squares
                         (integers-between 1 n)))))
```



Without the Abstractions

```
(define (sum-of-odd-squares n)
  (if (= n 1)
      1
      (let ((sq (square n)))
        (if (odd? sq)
            (+ sq (sum-of-odd-squares (- n 1)))
            (sum-of-odd-squares (- n 1))))))
```

"Spaghetti Code": unclear, easy to have bugs

but

Faster!

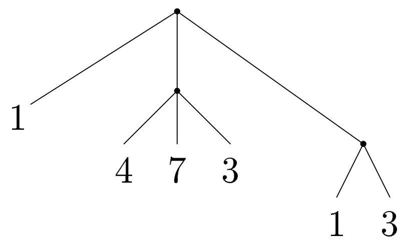
6.001 Sermon: Clearer is better

Optimize later—or leave optimization to the compiler.

Trees

1. A *leaf* of type T is an element of type T.
2. A Tree of type T's is either
   (a) a leaf of Type T, or
   (b) a list of Trees of type T's.

$$
\begin{aligned}
\mathrm{Tree}(T) &= \mathrm{Leaf}(T) \uplus (\mathrm{List}(\mathrm{Tree}(T))) \\
\mathrm{Leaf}(T) &= T
\end{aligned}
$$




```
(define test-tree (list 1
                        (list 4 7 3)
                        (list
                         (list 1 3))))
```

$$\texttt{test-tree} \Longrightarrow \underbrace{\texttt{(1 (4 7 3) ((1 3)))}}_{\text{output}}$$

Scheme Does Not Support Trees (We have to make our own)

For example,

```
(define (leaf? tr) (not (pair? tr)))

(define (make-leaf el) el)              ; el not a pair

(define (make-tree lst-of-trees) lst-of-trees)
```

(Silly now, but gets serious when trees have labels.)

Tree Contract follows from List Contract.

`Tree-map` : apply procedure to each leaf:

$$\texttt{test-tree} \Longrightarrow \underbrace{\texttt{(1 (4 7 3) ((1 3)))}}_{\text{output}}$$

`(tree-map square test-tree)` $\Longrightarrow$ `(1 (16 49 9) ((1 9)))`

```
(define (tree-map proc tr)
  (if (leaf? tr)
      (proc tr)
      (map (lambda (subtree) (tree-map proc subtree))
           tr)))
```

```
(define (flatten tree)
  (if (leaf? tree)
      (list tree)
      (accumulate append nil
                  (map flatten tree))))

(define (count-leaves tree)
  (if (leaf? tree)
      1
      (accumulate + 0
                  (map count-leaves tree))))
```

Abstraction: `tree-accumulate` (left-to-reader)