Built-in Common Lisp Functions for Creating Lists: CONS, APPEND, and LIST

- If $l \Rightarrow$ a proper list of length n
  then (CONS x l) $\Rightarrow$ a proper list of length n+1 obtained by **inserting the value of x at the *beginning* of the list of length n.**

  Examples:

-
-
-

.

-

- If l ⇒ a proper list of length n
  then (CONS x l) ⇒ a proper list of length n+1
  obtained by **inserting the value of x at the *beginning* of the list of length n.**

  Examples: (CONS 'A '(B C D)) ⇒
  (CONS (+ 2 3) '(A B C)) ⇒
  (CONS '(+ 2 3) '(A B C)) ⇒
  (CONS 'A nil) ⇒

-
-
-

.

-

- If *l* ⇒ a proper list of length *n*
  then (CONS *x* *l*) ⇒ a proper list of length *n*+1
  obtained by **inserting the value of *x* at the *beginning* of the list of length *n*.**

  Examples: (CONS 'A '(B C D)) ⇒ (A B C D)
  (CONS (+ 2 3) '(A B C)) ⇒
  (CONS '(+ 2 3) '(A B C)) ⇒
  (CONS 'A nil) ⇒

-
-
-

.

-

- If *l* ⇒ a proper list of length *n*
  then (CONS *x* *l*) ⇒ a proper list of length *n*+1
  obtained by **inserting the value of *x* at the *beginning* of the list of length *n*.**

  Examples: (CONS 'A '(B C D)) ⇒ (A B C D)
  (CONS (+ 2 3) '(A B C)) ⇒ (5 A B C)
  (CONS '(+ 2 3) '(A B C)) ⇒
  (CONS 'A nil) ⇒

-
-
-

.

-

- If *l* ⇒ a proper list of length *n*
  then (CONS *x* *l*) ⇒ a proper list of length *n*+1
  obtained by **inserting the value of *x* at the beginning of the list of length *n*.**

  Examples: (CONS 'A '(B C D)) ⇒ (A B C D)
  (CONS (+ 2 3) '(A B C)) ⇒ (5 A B C)
  (CONS '(+ 2 3) '(A B C)) ⇒ ((+ 2 3) A B C)
  (CONS 'A nil) ⇒

-
-
-

-

- If *l* ⇒ a proper list of length *n*
  then (CONS *x* *l*) ⇒ a proper list of length *n*+1
  obtained by **inserting the value of *x* at the *beginning* of the list of length *n*.**

  Examples:

  ```
  (CONS 'A '(B C D)) ⇒ (A B C D)
  (CONS (+ 2 3) '(A B C)) ⇒ (5 A B C)
  (CONS '(+ 2 3) '(A B C)) ⇒ ((+ 2 3) A B C)
  (CONS 'A nil) ⇒ (A)
  ```

-
-
-
-

- If *l* ⇒ a proper list of length *n*
  then (CONS *x* *l*) ⇒ a proper list of length *n*+1
  obtained by **inserting the value of *x* at the *beginning* of the list of length *n*.**

  Examples: (CONS 'A '(B C D)) ⇒ (A B C D)
  (CONS (+ 2 3) '(A B C)) ⇒ (5 A B C)
  (CONS '(+ 2 3) '(A B C)) ⇒ ((+ 2 3) A B C)
  (CONS 'A nil) ⇒ (A)
- (CAR (CONS x *l*)) ⇒ the value of
-
-
-

- If *l* ⇒ a proper list of length *n*
  then (CONS *x* *l*) ⇒ a proper list of length *n*+1
  obtained by **inserting the value of** *x* **at the *beginning* of the list of length** ***n*.**

  Examples:
  ```
  (CONS 'A '(B C D)) ⇒ (A B C D)
  (CONS (+ 2 3) '(A B C)) ⇒ (5 A B C)
  (CONS '(+ 2 3) '(A B C)) ⇒ ((+ 2 3) A B C)
  (CONS 'A nil) ⇒ (A)
  ```
- (CAR (CONS x *l*)) ⇒ the value of ***x***
-
-
-

- If *l* ⇒ a proper list of length *n*
  then (CONS *x* *l*) ⇒ a proper list of length *n*+1
  obtained by **inserting the value of** *x* **at the _beginning_ of the list of length** *n*.

  Examples: (CONS 'A '(B C D)) ⇒ (A B C D)
  (CONS (+ 2 3) '(A B C)) ⇒ (5 A B C)
  (CONS '(+ 2 3) '(A B C)) ⇒ ((+ 2 3) A B C)
  (CONS 'A nil) ⇒ (A)
- (CAR (CONS x *l*)) ⇒ the value of *x*
- (CDR (CONS x *l*)) ⇒ the value of
-
-

- If *l* ⇒ a proper list of length *n*
  then (CONS *x* *l*) ⇒ a proper list of length *n*+1
  obtained by **inserting the value of** ***x*** **at the** ***beginning*** **of the list of length** ***n*****.**

  Examples: (CONS 'A '(B C D)) ⇒ (A B C D)
  (CONS (+ 2 3) '(A B C)) ⇒ (5 A B C)
  (CONS '(+ 2 3) '(A B C)) ⇒ ((+ 2 3) A B C)
  (CONS 'A nil) ⇒ (A)
- (CAR (CONS x *l*)) ⇒ the value of *x*
- (CDR (CONS x *l*)) ⇒ the value of ***l***
-

.
-

- If *l* ⇒ a proper list of length *n*
  then (CONS *x* *l*) ⇒ a proper list of length *n*+1
  obtained by **inserting the value of *x* at the *<u>beginning</u>* of the list of length *n*.**

  Examples: (CONS 'A '(B C D)) ⇒ (A B C D)
  (CONS (+ 2 3) '(A B C)) ⇒ (5 A B C)
  (CONS '(+ 2 3) '(A B C)) ⇒ ((+ 2 3) A B C)
  (CONS 'A nil) ⇒ (A)
- (CAR (CONS x *l*)) ⇒ the value of *x*
- (CDR (CONS x *l*)) ⇒ the value of *l*
- A call of CONS must have ***<u>exactly two</u>*** arguments:
  Otherwise there will be an evaluation error.
-

- If *l* ⇒ a proper list of length *n*
  then (CONS *x* *l*) ⇒ a proper list of length *n*+1
  obtained by **inserting the value of** *x* **at the** ***<u>beginning</u>*** **of the list of length** *n***.**

  Examples:
  (CONS 'A '(B C D)) ⇒ (A B C D)
  (CONS (+ 2 3) '(A B C)) ⇒ (5 A B C)
  (CONS '(+ 2 3) '(A B C)) ⇒ ((+ 2 3) A B C)
  (CONS 'A nil) ⇒ (A)

- (CAR (CONS x *l*)) ⇒ the value of *x*
- (CDR (CONS x *l*)) ⇒ the value of *l*

- A call of CONS must have ***<u>exactly two</u>*** arguments: Otherwise there will be an evaluation error.

- If the 2nd argument value passed to CONS is ***<u>not</u>*** a proper list (i.e., if it is an atom other than NIL or a dotted list), then CONS returns a ***<u>dotted</u>*** list. But in this course you are ***<u>not</u>*** expected to call CONS this way!

- If *l* ⇒ a proper list of length *n*
  then (CONS *x* *l*) ⇒ a proper list of length *n*+1
  obtained by **inserting the value of *x* at the *beginning* of the list of length *n*.**

  Examples: (CONS 'A '(B C D)) ⇒ (A B C D)

- (CAR (CONS x *l*)) ⇒ the value of *x*
- (CDR (CONS x *l*)) ⇒ the value of *l*

- A call of CONS must have ***exactly two*** arguments: Otherwise there will be an evaluation error.

- If the 2nd argument value passed to CONS is ***not*** a proper list (i.e., if it is an atom other than NIL or a dotted list), then CONS returns a ***dotted*** list. But in this course you are ***not*** expected to call CONS this way!

- If *l* ⇒ a proper list of length *n*
  then (CONS *x l*) ⇒ a proper list of length *n*+1
  obtained by **inserting the value of** *x* **at the *beginning* of the list of length** *n***.**

  Examples: (CONS 'A '(B C D)) ⇒ (A B C D)

- (CAR (CONS x *l*)) ⇒ the value of *x*
- (CDR (CONS x *l*)) ⇒ the value of *l*
- A call of CONS must have ***exactly two*** arguments: Otherwise there will be an evaluation error.

- If the 2nd argument value passed to CONS is ***not*** a proper list (i.e., if it is an atom other than NIL or a dotted list), then CONS returns a ***dotted*** list. But in this course you are ***not*** expected to call CONS this way!

- If *l* ⇒ a proper list of length *n*
  then (CONS *x* *l*) ⇒ a proper list of length *n*+1 obtained by **inserting the value of *x* at the *beginning* of the list of length *n*.**

  Examples: (CONS 'A '(B C D)) ⇒ (A B C D)

- (CAR (CONS x *l*)) ⇒ the value of *x*
- (CDR (CONS x *l*)) ⇒ the value of *l*

- A call of CONS must have ***exactly two*** arguments: Otherwise there will be an evaluation error.

- If the 2nd argument value passed to CONS is ***not*** a proper list (i.e., if it is an atom other than NIL or a dotted list), then CONS returns a ***dotted*** list. But in this course you are ***not*** expected to call CONS this way!

The name "CONS" is a shortened form of "construct":
  - CONS is the most basic way to construct a new proper list; but the CDR of that new list will be an existing list.

- If l_1 ⇒ a proper list of length n_1
  and l_2 ⇒ a proper list of length n_2
  then (**APPEND** l_1 l_2) ⇒ a proper list of length $n_1 + n_2$
  obtained by **concatenating the lists given by l_1 and l_2.**

  EXAMPLES:

- If l_1 ⇒ a proper list of length n_1
  and l_2 ⇒ a proper list of length n_2
  then (**APPEND** l_1 l_2) ⇒ a proper list of length $n_1 + n_2$ obtained by **concatenating the lists given by l_1 and l_2.**

  EXAMPLES: (APPEND '(A B C) '(D E)) =>
  (APPEND '((A B) C) '(D (E))) =>
  (APPEND NIL '(A B C)) =>

- If $l_1 \Rightarrow$ a proper list of length n_1
  and $l_2 \Rightarrow$ a proper list of length n_2
  then (**APPEND** l_1 l_2) $\Rightarrow$ a proper list of length $n_1 + n_2$
  obtained by **concatenating the lists given by l_1 and l_2.**

  EXAMPLES:
```
(APPEND '(A B C) '(D E)) => (A B C D E)
(APPEND '((A B) C) '(D (E))) =>
(APPEND NIL '(A B C)) =>
```

- If $l_1 \Rightarrow$ a proper list of length n_1
  and $l_2 \Rightarrow$ a proper list of length n_2
  then (**APPEND** l_1 l_2) $\Rightarrow$ a proper list of length $n_1 + n_2$ obtained by **concatenating the lists given by l_1 and l_2.**

  EXAMPLES:

```
(APPEND '(A B C) '(D E)) => (A B C D E)
(APPEND '((A B) C) '(D (E))) => ((A B) C D (E))
(APPEND NIL '(A B C)) =>
```

- If $l_1 \Rightarrow$ a proper list of length n_1
  and $l_2 \Rightarrow$ a proper list of length n_2
  then (**APPEND** l_1 l_2) $\Rightarrow$ a proper list of length $n_1 + n_2$
  obtained by **concatenating the lists given by l_1 and l_2.**

  EXAMPLES:

```
(APPEND '(A B C) '(D E)) => (A B C D E)
(APPEND '((A B) C) '(D (E))) => ((A B) C D (E))
(APPEND NIL '(A B C)) => (A B C)
```

- If $l_1 \Rightarrow$ a proper list of length n_1
  and $l_2 \Rightarrow$ a proper list of length n_2
  then (**APPEND** l_1 l_2) $\Rightarrow$ a proper list of length $n_1 + n_2$ obtained by **concatenating the lists given by l_1 and l_2.**

  EXAMPLES:
  ```
  (APPEND '(A B C) '(D E)) => (A B C D E)
  (APPEND '((A B) C) '(D (E))) => ((A B) C D (E))
  (APPEND NIL '(A B C)) => (A B C)
  ```

More generally:

- If $l_1, \ldots, l_k \Rightarrow k$ proper lists of lengths $n_1, \ldots, n_k$, then (APPEND $l_1 \ldots l_k$) $\Rightarrow$ a proper list of length $n_1 + \ldots + n_k$ obtained by **concatenating those k lists.**

  EXAMPLE:

- If $l_1 \Rightarrow$ a proper list of length n_1
  and $l_2 \Rightarrow$ a proper list of length n_2
  then (**APPEND** l_1 l_2) $\Rightarrow$ a proper list of length $n_1 + n_2$ obtained by **concatenating the lists given by l_1 and l_2.**

  EXAMPLES:
  ```
  (APPEND '(A B C) '(D E)) => (A B C D E)
  (APPEND '((A B) C) '(D (E))) => ((A B) C D (E))
  (APPEND NIL '(A B C)) => (A B C)
  ```

More generally:

- If l_1, …, $l_k \Rightarrow k$ proper lists of lengths $n_1, \ldots, n_k$, then (APPEND l_1 … l_k) $\Rightarrow$ a proper list of length $n_1 + \ldots + n_k$ obtained by **concatenating those *k* lists.**

  EXAMPLE:
  ```
  (APPEND '(A B) '(C D E) '(F)) =>
  ```

- If $l_1 \Rightarrow$ a proper list of length n_1
  and $l_2 \Rightarrow$ a proper list of length n_2
  then (**APPEND** l_1 l_2) $\Rightarrow$ a proper list of length $n_1 + n_2$ obtained by **concatenating the lists given by l_1 and l_2.**

  EXAMPLES:
  ```
  (APPEND '(A B C) '(D E)) => (A B C D E)
  (APPEND '((A B) C) '(D (E))) => ((A B) C D (E))
  (APPEND NIL '(A B C)) => (A B C)
  ```

More generally:

- If l_1, …, $l_k \Rightarrow k$ proper lists of lengths $n_1, \ldots, n_k$, then (APPEND l_1 … l_k) $\Rightarrow$ a proper list of length $n_1 + \ldots + n_k$ obtained by **concatenating those *k* lists.**

  EXAMPLE:
  ```
  (APPEND '(A B) '(C D E) '(F)) => (A B C D E F)
  ```

- If $l_1 \Rightarrow$ a proper list of length n_1
  and $l_2 \Rightarrow$ a proper list of length n_2
  then (**APPEND** l_1 l_2) $\Rightarrow$ a proper list of length $n_1 + n_2$ obtained by **concatenating the lists given by l_1 and l_2.**

- If l_1, …, $l_k \Rightarrow k$ proper lists of lengths $n_1, \ldots, n_k$, then (APPEND l_1 … l_k) $\Rightarrow$ a proper list of length $n_1 + \ldots + n_k$ obtained by **concatenating those *k* lists.**

  EXAMPLE: (APPEND '(A B) '(C D E) '(F)) => (A B C D E F)

- If $l_1 \Rightarrow$ a proper list of length n_1
  and $l_2 \Rightarrow$ a proper list of length n_2
  then (**APPEND** l_1 l_2) $\Rightarrow$ a proper list of length $n_1 + n_2$
  obtained by **concatenating the lists given by l_1 and l_2.**
- If l_1, …, $l_k \Rightarrow k$ proper lists of lengths $n_1, \ldots, n_k$, then
  (APPEND l_1 … l_k) $\Rightarrow$ a proper list of length $n_1 + \ldots + n_k$
  obtained by **concatenating those *k* lists.**

- If $l_1 \Rightarrow$ a proper list of length n_1
  and $l_2 \Rightarrow$ a proper list of length n_2
  then (APPEND l_1 l_2) $\Rightarrow$ a proper list of length $n_1 + n_2$
  obtained by **concatenating the lists given by l_1 and l_2.**
- If l_1, …, $l_k \Rightarrow$ k proper lists of lengths $n_1, \ldots, n_k$, then
  (APPEND l_1 … l_k) $\Rightarrow$ a proper list of length $n_1 + \ldots + n_k$
  obtained by **concatenating those k lists.**

- If $l_1 \Rightarrow$ a proper list of length n_1
  and $l_2 \Rightarrow$ a proper list of length n_2
  then (APPEND l_1 l_2) $\Rightarrow$ a proper list of length $n_1 + n_2$
  obtained by **concatenating the lists given by l_1 and l_2.**
- If l_1, …, $l_k \Rightarrow$ k proper lists of lengths $n_1, \ldots, n_k$, then
  (APPEND l_1 … l_k) $\Rightarrow$ a proper list of length $n_1 + \ldots + n_k$
  obtained by **concatenating those k lists.**

When APPEND is called with $k \geq 2$ arguments:

-

-

- If $l_1 \Rightarrow$ a proper list of length n_1
  and $l_2 \Rightarrow$ a proper list of length n_2
  then (APPEND l_1 l_2) $\Rightarrow$ a proper list of length $n_1 + n_2$
  obtained by **concatenating the lists given by l_1 and l_2.**
- If $l_1, \ldots, l_k \Rightarrow k$ proper lists of lengths $n_1, \ldots, n_k$, then
  (APPEND $l_1 \ldots l_k$) $\Rightarrow$ a proper list of length $n_1 + \ldots + n_k$
  obtained by **concatenating those k lists.**

When APPEND is called with $k \geq 2$ arguments:

- If any of the first k-1 argument values is ***not*** a proper list, then there will be an evaluation error.

  EXAMPLE:

-

- If $l_1 \Rightarrow$ a proper list of length n_1
  and $l_2 \Rightarrow$ a proper list of length n_2
  then (APPEND l_1 l_2) $\Rightarrow$ a proper list of length $n_1 + n_2$
  obtained by **concatenating the lists given by l_1 and l_2.**
- If $l_1, \ldots, l_k \Rightarrow k$ proper lists of lengths $n_1, \ldots, n_k$, then
  (APPEND $l_1 \ldots l_k$) $\Rightarrow$ a proper list of length $n_1 + \ldots + n_k$
  obtained by **concatenating those *k* lists.**

When APPEND is called with $k \geq 2$ arguments:

- If any of the first k-1 argument values is ***not*** a proper list, then there will be an evaluation error.

  EXAMPLE: Evaluation of (APPEND (+ 3 4) '(C D E)) produces an error, because 7 is not a list.
-

- If $l_1 \Rightarrow$ a proper list of length n_1
  and $l_2 \Rightarrow$ a proper list of length n_2
  then (APPEND l_1 l_2) $\Rightarrow$ a proper list of length $n_1 + n_2$
  obtained by **concatenating the lists given by l_1 and l_2.**
- If $l_1, \ldots, l_k \Rightarrow k$ proper lists of lengths $n_1, \ldots, n_k$, then
  (APPEND $l_1 \ldots l_k$) $\Rightarrow$ a proper list of length $n_1 + \ldots + n_k$
  obtained by **concatenating those k lists.**

When APPEND is called with $k \geq 2$ arguments:

- If any of the first k-1 argument values is ***not*** a proper list, then there will be an evaluation error.

  EXAMPLE: Evaluation of (APPEND (+ 3 4) '(C D E)) produces an error, because 7 is not a list.

- If the first k-1 argument values are proper lists but the k^{th} argument value is not, then APPEND returns a *dotted* list (unless the first k-1 argument values are all NIL). But in this course you are ***not*** expected to call APPEND this way!

(LIST e_1 … e_k) $\Rightarrow$ a proper list of length k whose i^{th} element ($1 \leq i \leq k$) is the value of e_i.

EXAMPLES:

(LIST e_1 … e_k) ⇒ a proper list of length k whose i^{th} element ($1 \le i \le k$) is the value of e_i.

EXAMPLES:

```
(LIST 'P '(A B) (+ 3 4) '(F)) ⇒
(LIST (CAR '(A B)) 6 (CDR '(X Y))) ⇒
(LIST '(U V W)) ⇒
```

(LIST e_1 … e_k) ⇒ a proper list of length k whose i^{th} element ($1 \le i \le k$) is the value of e_i.

EXAMPLES:

```
(LIST 'P '(A B) (+ 3 4) '(F)) ⇒ (P (A B) 7 (F))
(LIST (CAR '(A B)) 6 (CDR '(X Y))) ⇒
(LIST '(U V W)) ⇒
```

(LIST e_1 … e_k) ⇒ a proper list of length k whose i^{th} element ($1 \le i \le k$) is the value of e_i.

EXAMPLES:

```
(LIST 'P '(A B) (+ 3 4) '(F)) ⇒ (P (A B) 7 (F))
(LIST (CAR '(A B)) 6 (CDR '(X Y))) ⇒ (A 6 (Y))
(LIST '(U V W)) ⇒
```

(LIST e_1 … e_k) ⇒ a proper list of length k whose
i^{th} element ($1 \le i \le k$) is the
value of e_i.

EXAMPLES:

```
(LIST 'P '(A B) (+ 3 4) '(F)) ⇒ (P (A B) 7 (F))
(LIST (CAR '(A B)) 6 (CDR '(X Y))) ⇒ (A 6 (Y))
(LIST '(U V W)) ⇒ ((U V W))
```

(LIST e_1 … e_k) ⇒ a proper list of length k whose i^{th} element ($1 \le i \le k$) is the value of e_i.

EXAMPLES:

```
(LIST 'P '(A B) (+ 3 4) '(F)) ⇒ (P (A B) 7 (F))
(LIST (CAR '(A B)) 6 (CDR '(X Y))) ⇒ (A 6 (Y))
(LIST '(U V W)) ⇒ ((U V W))
```

Don't confuse the functions CONS, APPEND, and LIST!

```
(CONS                    ) ⇒
(APPEND                    ) ⇒
(LIST                    ) ⇒
```

(LIST e_1 … e_k) ⇒ a proper list of length k whose i^{th} element ($1 \le i \le k$) is the value of e_i.

EXAMPLES:
```
(LIST 'P '(A B) (+ 3 4) '(F)) ⇒ (P (A B) 7 (F))
(LIST (CAR '(A B)) 6 (CDR '(X Y))) ⇒ (A 6 (Y))
(LIST '(U V W)) ⇒ ((U V W))
```

Don't confuse the functions CONS, APPEND, and LIST.
```
(CONS '(1 2 3) '(4 5)) ⇒
(APPEND '(1 2 3) '(4 5)) ⇒
(LIST '(1 2 3) '(4 5)) ⇒
```

(LIST e_1 … e_k) ⇒ a proper list of length k whose i^{th} element ($1 \leq i \leq k$) is the value of e_i.

```
EXAMPLES:  (LIST 'P '(A B) (+ 3 4) '(F)) ⇒ (P (A B) 7 (F))
           (LIST (CAR '(A B)) 6 (CDR '(X Y))) ⇒ (A 6 (Y))
           (LIST '(U V W)) ⇒ ((U V W))
```

Don't confuse the functions CONS, APPEND, and LIST.

```
      (CONS '(1 2 3) '(4 5)) ⇒ ((1 2 3) 4 5)
      (APPEND '(1 2 3) '(4 5)) ⇒
      (LIST '(1 2 3) '(4 5)) ⇒
```

(LIST e_1 … e_k) ⇒ a proper list of length k whose i^{th} element ($1 \le i \le k$) is the value of e_i.

```
EXAMPLES:  (LIST 'P '(A B) (+ 3 4) '(F)) ⇒ (P (A B) 7 (F))
           (LIST (CAR '(A B)) 6 (CDR '(X Y))) ⇒ (A 6 (Y))
           (LIST '(U V W)) ⇒ ((U V W))
```

Don't confuse the functions CONS, APPEND, and LIST.

```
(CONS '(1 2 3) '(4 5)) ⇒ ((1 2 3) 4 5)
(APPEND '(1 2 3) '(4 5)) ⇒ (1 2 3 4 5)
(LIST '(1 2 3) '(4 5)) ⇒
```

(LIST e_1 … e_k) ⇒ a proper list of length k whose i^{th} element ($1 \le i \le k$) is the value of e_i.

```
EXAMPLES:  (LIST 'P '(A B) (+ 3 4) '(F)) ⇒ (P (A B) 7 (F))
           (LIST (CAR '(A B)) 6 (CDR '(X Y))) ⇒ (A 6 (Y))
           (LIST '(U V W)) ⇒ ((U V W))
```

Don't confuse the functions CONS, APPEND, and LIST.

```
(CONS '(1 2 3) '(4 5)) ⇒ ((1 2 3) 4 5)
(APPEND '(1 2 3) '(4 5)) ⇒ (1 2 3 4 5)
(LIST '(1 2 3) '(4 5)) ⇒ ((1 2 3) (4 5))
```

```
(LIST e1 … ek) ⇒ a proper list of length k whose
                 ith element (1 ≤ i ≤ k) is the
                 value of ei.
```

$(\texttt{LIST}\ e_1\ \ldots\ e_k) \Rightarrow$ a proper list of length k whose i^{th} element $(1 \le i \le k)$ is the value of e_i.

EXAMPLES:

```
(LIST 'P '(A B) (+ 3 4) '(F)) ⇒ (P (A B) 7 (F))
(LIST (CAR '(A B)) 6 (CDR '(X Y))) ⇒ (A 6 (Y))
(LIST '(U V W)) ⇒ ((U V W))
```

Don't confuse the functions CONS, APPEND, and LIST.

```
(CONS '(1 2 3) '(4 5)) ⇒ ((1 2 3) 4 5)
(APPEND '(1 2 3) '(4 5)) ⇒ (1 2 3 4 5)
(LIST '(1 2 3) '(4 5)) ⇒ ((1 2 3) (4 5))
```

•

•

•

(LIST e_1 … e_k) ⇒ a proper list of length k whose i^{th} element ($1 \le i \le k$) is the value of e_i.

EXAMPLES:

```
(LIST 'P '(A B) (+ 3 4) '(F)) ⇒ (P (A B) 7 (F))
(LIST (CAR '(A B)) 6 (CDR '(X Y))) ⇒ (A 6 (Y))
(LIST '(U V W)) ⇒ ((U V W))
```

Don't confuse the functions CONS, APPEND, and LIST.

```
(CONS '(1 2 3) '(4 5)) ⇒ ((1 2 3) 4 5)
(APPEND '(1 2 3) '(4 5)) ⇒ (1 2 3 4 5)
(LIST '(1 2 3) '(4 5)) ⇒ ((1 2 3) (4 5))
```

- (LIST *e*) = (CONS)
-
-

(LIST e_1 … e_k) ⇒ a proper list of length k whose i^{th} element ($1 \le i \le k$) is the value of e_i.

EXAMPLES: (LIST 'P '(A B) (+ 3 4) '(F)) ⇒ (P (A B) 7 (F))
(LIST (CAR '(A B)) 6 (CDR '(X Y))) ⇒ (A 6 (Y))
(LIST '(U V W)) ⇒ ((U V W))

Don't confuse the functions CONS, APPEND, and LIST.

(CONS '(1 2 3) '(4 5)) ⇒ ((1 2 3) 4 5)
(APPEND '(1 2 3) '(4 5)) ⇒ (1 2 3 4 5)
(LIST '(1 2 3) '(4 5)) ⇒ ((1 2 3) (4 5))

- (LIST *e*) = (CONS *e* NIL)
-
-

(LIST e_1 … e_k) ⇒ a proper list of length k whose i^{th} element ($1 \le i \le k$) is the value of e_i.

EXAMPLES:

```
(LIST 'P '(A B) (+ 3 4) '(F)) ⇒ (P (A B) 7 (F))
(LIST (CAR '(A B)) 6 (CDR '(X Y))) ⇒ (A 6 (Y))
(LIST '(U V W)) ⇒ ((U V W))
```

Don't confuse the functions CONS, APPEND, and LIST.

```
(CONS '(1 2 3) '(4 5)) ⇒ ((1 2 3) 4 5)
(APPEND '(1 2 3) '(4 5)) ⇒ (1 2 3 4 5)
(LIST '(1 2 3) '(4 5)) ⇒ ((1 2 3) (4 5))
```

- (LIST *e*) = (CONS *e* NIL)
- (CONS *e* *l*) = (APPEND)
-

(LIST e_1 … e_k) ⇒ a proper list of length k whose i^{th} element ($1 \le i \le k$) is the value of e_i.

```
EXAMPLES:  (LIST 'P '(A B) (+ 3 4) '(F)) ⇒ (P (A B) 7 (F))
           (LIST (CAR '(A B)) 6 (CDR '(X Y))) ⇒ (A 6 (Y))
           (LIST '(U V W)) ⇒ ((U V W))
```

Don't confuse the functions CONS, APPEND, and LIST.

```
(CONS '(1 2 3) '(4 5)) ⇒ ((1 2 3) 4 5)
(APPEND '(1 2 3) '(4 5)) ⇒ (1 2 3 4 5)
(LIST '(1 2 3) '(4 5)) ⇒ ((1 2 3) (4 5))
```

- (LIST *e*) = (CONS *e* NIL)
- (CONS *e* *l*) = (APPEND (LIST *e*) *l*)
-

(LIST e_1 … e_k) ⇒ a proper list of length k whose i^{th} element ($1 \le i \le k$) is the value of e_i.

EXAMPLES: (LIST 'P '(A B) (+ 3 4) '(F)) ⇒ (P (A B) 7 (F))
(LIST (CAR '(A B)) 6 (CDR '(X Y))) ⇒ (A 6 (Y))
(LIST '(U V W)) ⇒ ((U V W))

Don't confuse the functions CONS, APPEND, and LIST.

```
(CONS '(1 2 3) '(4 5)) ⇒ ((1 2 3) 4 5)
(APPEND '(1 2 3) '(4 5)) ⇒ (1 2 3 4 5)
(LIST '(1 2 3) '(4 5)) ⇒ ((1 2 3) (4 5))
```

- (LIST *e*) = (CONS *e* NIL)
- (CONS *e* *l*) = (APPEND (LIST *e*) *l*)
- Evaluation of (CONS *e* *l*), (APPEND l_1 … l_k), and (LIST e_1 … e_k) does ***not*** change the values of the *e*'s and the *l*'s.
  -

(LIST e_1 … e_k) ⇒ a proper list of length k whose i^{th} element ($1 \le i \le k$) is the value of e_i.

EXAMPLES:

```
(LIST 'P '(A B) (+ 3 4) '(F)) ⇒ (P (A B) 7 (F))
(LIST (CAR '(A B)) 6 (CDR '(X Y))) ⇒ (A 6 (Y))
(LIST '(U V W)) ⇒ ((U V W))
```

Don't confuse the functions CONS, APPEND, and LIST.

```
(CONS '(1 2 3) '(4 5)) ⇒ ((1 2 3) 4 5)
(APPEND '(1 2 3) '(4 5)) ⇒ (1 2 3 4 5)
(LIST '(1 2 3) '(4 5)) ⇒ ((1 2 3) (4 5))
```

- (LIST e) = (CONS e NIL)
- (CONS e l) = (APPEND (LIST e) l)
- Evaluation of (CONS e l), (APPEND l_1 … l_k), and (LIST e_1 … e_k) does ***not*** change the values of the e's and the l's.
  - If this were not the case, we would not be able to use these functions in functional programming!

More Built-in Functions That Extract Parts of Lists: SECOND, ..., TENTH, and C … R Functions

SECOND, THIRD, etc.

Recall: If l ⇒ a nonempty list, then
(FIRST l) = (CAR l) evaluates to the
1st element of the list given by l.

Also, (FIRST NIL) = (CAR NIL) ⇒ NIL.

Similarly:

-

-

-

SECOND, THIRD, etc.

Recall: If l ⇒ a nonempty list, then
(FIRST l) = (CAR l) evaluates to the
<u>1st</u> element of the list given by l.

Also, (FIRST NIL) = (CAR NIL) ⇒ NIL.

Similarly:

- If l ⇒ a list of length ≥ 2, then
(SECOND l) = (CAR (CDR l)) evaluates to
the **<u>2nd</u>** element of the list given by l.
-
-

SECOND, THIRD, etc.

Recall: If $l \Rightarrow$ a nonempty list, then
(FIRST l) = (CAR l) evaluates to the
<u>1st</u> element of the list given by l.

Also, (FIRST NIL) = (CAR NIL) $\Rightarrow$ NIL.

Similarly:

- If $l \Rightarrow$ a list of length ≥ 2, then
(SECOND l) = (CAR (CDR l)) evaluates to
the **<u>2nd</u>** element of the list given by l.

  (SECOND l) $\Rightarrow$ NIL if $l \Rightarrow$ a proper list of length ≤ 1.

-

-

SECOND, THIRD, etc.

Recall: If l ⇒ a nonempty list, then (FIRST l) = (CAR l) evaluates to the **<u>1st</u>** element of the list given by l.

Also, (FIRST NIL) = (CAR NIL) ⇒ NIL.

Similarly:

- If l ⇒ a list of length ≥ 2, then **(SECOND l)** = (CAR (CDR l)) evaluates to the **<u>2nd</u>** element of the list given by l.

  (SECOND l) ⇒ NIL if l ⇒ a proper list of length ≤ 1.
- If l ⇒ a list of length ≥ 3, then **(THIRD l)** = (CAR (CDR (CDR l))) evaluates to the **<u>3rd</u>** element of the list given by l.
-

SECOND, THIRD, etc.

Recall: If $l \Rightarrow$ a nonempty list, then (FIRST l) = (CAR l) evaluates to the <u>**1st**</u> element of the list given by l.

Also, (FIRST NIL) = (CAR NIL) $\Rightarrow$ NIL.

Similarly:

- If $l \Rightarrow$ a list of length ≥ 2, then **(SECOND l)** = (CAR (CDR l)) evaluates to the <u>**2nd**</u> element of the list given by l.

  (SECOND l) $\Rightarrow$ NIL if $l \Rightarrow$ a proper list of length ≤ 1.
- If $l \Rightarrow$ a list of length ≥ 3, then **(THIRD l)** = (CAR (CDR (CDR l))) evaluates to the <u>**3rd**</u> element of the list given by l.

  (THIRD l) $\Rightarrow$ NIL if $l \Rightarrow$ a proper list of length ≤ 2.
-

SECOND, THIRD, etc.

Recall: If l ⇒ a nonempty list, then (FIRST l) = (CAR l) evaluates to the **1st** element of the list given by l.

Also, (FIRST NIL) = (CAR NIL) ⇒ NIL.

Similarly:

- If l ⇒ a list of length ≥ 2, then **(SECOND l)** = (CAR (CDR l)) evaluates to the **2nd** element of the list given by l.

  (SECOND l) ⇒ NIL if l ⇒ a proper list of length ≤ 1.

- If l ⇒ a list of length ≥ 3, then **(THIRD l)** = (CAR (CDR (CDR l))) evaluates to the **3rd** element of the list given by l.

  (THIRD l) ⇒ NIL if l ⇒ a proper list of length ≤ 2.

- **(FOURTH l)**, ..., **(TENTH l)** are defined analogously.

C … R Functions

Each C … R function is equivalent to the **composition of a certain sequence of CARs and/or CDRs**:

-
-
-
-
-
-
-
-
-

C … R Functions

Each C … R function is equivalent to the **composition of a certain sequence of CARs and/or CDRs**:

- (C**AD**R *l*) = (C**A**R (C**D**R *l*)) = (SECOND *l*)
-
-
-
-
-
-
-
-

C … R Functions

Each C … R function is equivalent to the **composition of a certain sequence of CARs and/or CDRs**:

- (C**AD**R *l*) = (C**A**R (C**D**R *l*)) = (SECOND *l*)
- (C**ADD**R *l*) = (C**A**R (C**D**R (C**D**R *l*))) = (THIRD *l*)
-
-
-
-
-
-
-

C … R Functions

Each C … R function is equivalent to the **composition of a certain sequence of CARs and/or CDRs**:

- (C**AD**R *l*) = (C**A**R (C**D**R *l*)) = (SECOND *l*)
- (C**ADD**R *l*) = (C**A**R (C**D**R (C**D**R *l*))) = (THIRD *l*)
- (C**ADDD**R *l*) = (C**A**R (C**D**R (C**D**R (C**D**R *l*)))) = (FOURTH *l*)

C … R Functions

Each C … R function is equivalent to the **composition of a certain sequence of CARs and/or CDRs**:

- (C**AD**R *l*) = (C**A**R (C**D**R *l*)) = (SECOND *l*)
- (C**ADD**R *l*) = (C**A**R (C**D**R (C**D**R *l*))) = (THIRD *l*)
- (C**ADDD**R *l*) = (C**A**R (C**D**R (C**D**R (C**D**R *l*)))) = (FOURTH *l*)
- (C**AA**R *l*) = (C**A**R (C**A**R *l*)) = (FIRST (FIRST *l*))
-
-
-
-
-

C … R Functions

Each C … R function is equivalent to the **composition of a certain sequence of CARs and/or CDRs**:

- (C**AD**R *l*) = (C**A**R (C**D**R *l*)) = (SECOND *l*)
- (C**ADD**R *l*) = (C**A**R (C**D**R (C**D**R *l*))) = (THIRD *l*)
- (C**ADDD**R *l*) = (C**A**R (C**D**R (C**D**R (C**D**R *l*)))) = (FOURTH *l*)
- (C**AA**R *l*) = (C**A**R (C**A**R *l*)) = (FIRST (FIRST *l*))
- (C**DA**R *l*) = (C**D**R (C**A**R *l*)) = (REST (FIRST *l*))
-
-
-
-

C … R Functions

Each C … R function is equivalent to the **composition of a certain sequence of CARs and/or CDRs**:

- (C**AD**R *l*) = (C**A**R (C**D**R *l*)) = (SECOND *l*)
- (C**ADD**R *l*) = (C**A**R (C**D**R (C**D**R *l*))) = (THIRD *l*)
- (C**ADDD**R *l*) = (C**A**R (C**D**R (C**D**R (C**D**R *l*)))) = (FOURTH *l*)
- (C**AA**R *l*) = (C**A**R (C**A**R *l*)) = (FIRST (FIRST *l*))
- (C**DA**R *l*) = (C**D**R (C**A**R *l*)) = (REST (FIRST *l*))
- (C**DADD**R *l*) = (C**D**R (C**A**R (C**D**R (C**D**R *l*))))
= (C**D**R (C**ADD**R *l*)) = (REST (THIRD *l*))
-
-
-

C … R Functions

Each C … R function is equivalent to the **composition of a certain sequence of CARs and/or CDRs**:

- (CADR *l*) = (CAR (CDR *l*)) = (SECOND *l*)
- (CADDR *l*) = (CAR (CDR (CDR *l*))) = (THIRD *l*)
- (CADDDR *l*) = (CAR (CDR (CDR (CDR *l*)))) = (FOURTH *l*)
- (CAAR *l*) = (CAR (CAR *l*)) = (FIRST (FIRST *l*))
- (CDAR *l*) = (CDR (CAR *l*)) = (REST (FIRST *l*))
- (CDADDR *l*) = (CDR (CAR (CDR (CDR *l*))))
= (CDR (CADDR *l*)) = (REST (THIRD *l*))
- (CADADR *l*) = (CAR (CDR (CAR (CDR *l*))))
= (CADR (CADR *l*)) = (SECOND (SECOND *l*))
-
-

C … R Functions

Each C … R function is equivalent to the **composition of a certain sequence of CARs and/or CDRs**:

- (CADR *l*) = (CAR (CDR *l*)) = (SECOND *l*)
- (CADDR *l*) = (CAR (CDR (CDR *l*))) = (THIRD *l*)
- (CADDDR *l*) = (CAR (CDR (CDR (CDR *l*)))) = (FOURTH *l*)
- (CAAR *l*) = (CAR (CAR *l*)) = (FIRST (FIRST *l*))
- (CDAR *l*) = (CDR (CAR *l*)) = (REST (FIRST *l*))
- (CDADDR *l*) = (CDR (CAR (CDR (CDR *l*))))
  = (CDR (CADDR *l*)) = (REST (THIRD *l*))
- (CADADR *l*) = (CAR (CDR (CAR (CDR *l*))))
  = (CADR (CADR *l*)) = (SECOND (SECOND *l*))

and similarly for all other sequences of **2 – 4** **A**s and/or **D**s.

- Although authors sometimes write C … R function names that contain ***more than 4*** **A**s and/or **D**s, functions with such names are ***not*** built-in functions of most implementations of Common Lisp!
-

C … R Functions

Each C … R function is equivalent to the **composition of a certain sequence of CARs and/or CDRs**:

- (C**AD**R *l*) = (C**A**R (C**D**R *l*)) = (SECOND *l*)
- (C**ADD**R *l*) = (C**A**R (C**D**R (C**D**R *l*))) = (THIRD *l*)
- (C**ADDD**R *l*) = (C**A**R (C**D**R (C**D**R (C**D**R *l*)))) = (FOURTH *l*)
- (C**AA**R *l*) = (C**A**R (C**A**R *l*)) = (FIRST (FIRST *l*))
- (C**DA**R *l*) = (C**D**R (C**A**R *l*)) = (REST (FIRST *l*))
- (C**DADD**R *l*) = (C**D**R (C**A**R (C**D**R (C**D**R *l*))))
= (C**D**R (C**ADD**R *l*)) = (REST (THIRD *l*))
- (C**ADAD**R *l*) = (C**A**R (C**D**R (C**A**R (C**D**R *l*))))
= (C**AD**R (C**AD**R *l*)) = (SECOND (SECOND *l*))

and similarly for all other sequences of **2 – 4** **A**s and/or **D**s.

- Although authors sometimes write C … R function names that contain ***more than 4*** **A**s and/or **D**s, functions with such names are ***not*** built-in functions of most implementations of Common Lisp!
- C … R function names are pronounceable: This is one reason the nondescriptive names CAR and CDR are still used.

From p. 48 of Touretzky:

CAR/CDR Pronunciation Guide

Function	Pronunciation	Alternate Name
CAR	*kar*	FIRST
CDR	*cou-der*	REST
CAAR	*ka-ar*	
CADR	*kae-der*	SECOND
CDAR	*cou-dar*	
CDDR	*cou-dih-der*	
CAAAR	*ka-a-ar*	
CAADR	*ka-ae-der*	
CADAR	*ka-dar*	
CADDR	*ka-dih-der*	THIRD
CDAAR	*cou-da-ar*	
CDADR	*cou-dae-der*	
CDDAR	*cou-dih-dar*	
CDDDR	*cou-did-dih-der*	
CADDDR	*ka-dih-dih-der*	FOURTH

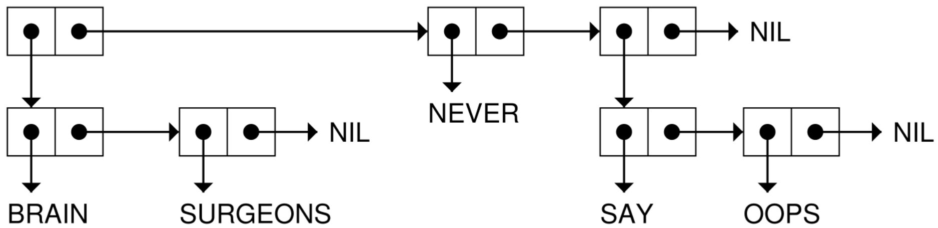
and so on

CONS Cells and the Implementation of CAR, CDR, and CONS

We have already seen "*box* and *arrow*" drawings of Lisp data objects represented by lists. Below is one example:

From p. 34 of Touretzky.

((BRAIN SURGEONS) NEVER (SAY OOPS)) represents:

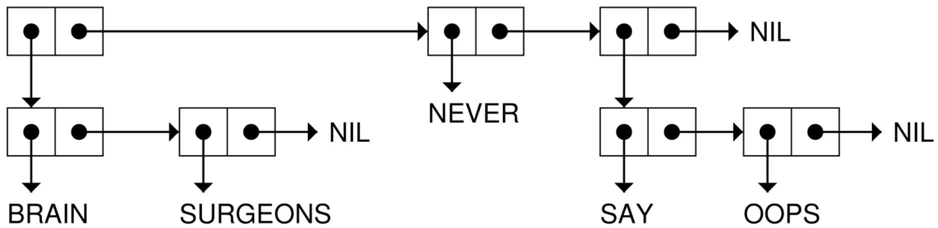


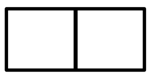
-
-

We have already seen "*box* and *arrow*" drawings of Lisp data objects represented by lists. Below is one example:

From p. 34 of Touretzky.

((BRAIN SURGEONS) NEVER (SAY OOPS)) represents:

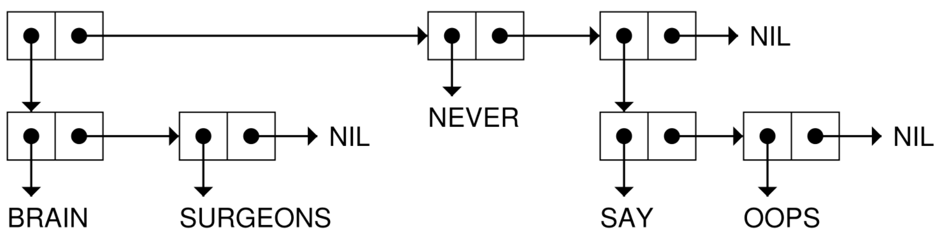


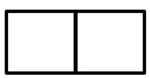
- Each of the  boxes in such a drawing represents a Lisp data object called a ***cons cell***.
-

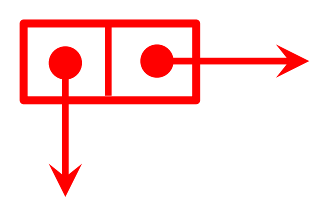
We have already seen "*box* and *arrow*" drawings of Lisp data objects represented by lists. Below is one example:

From p. 34 of Touretzky.

((BRAIN SURGEONS) NEVER (SAY OOPS)) represents:



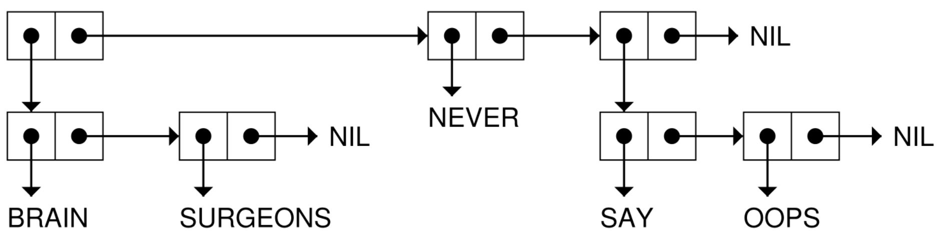
- Each of the  boxes in such a drawing represents a Lisp data object called a ***cons cell***.
- A cons cell has 2 fields that contain pointers. (Pointers are represented by the arrows in box and arrow drawings.)
  -
  -

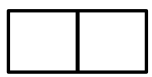


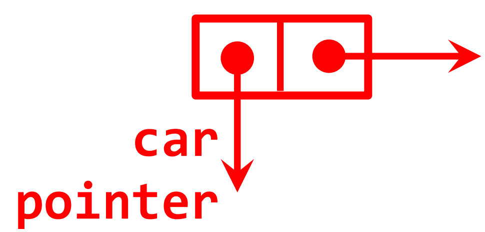
We have already seen "*box* and *arrow*" drawings of Lisp data objects represented by lists. Below is one example:

From p. 34 of Touretzky.

((BRAIN SURGEONS) NEVER (SAY OOPS)) represents:



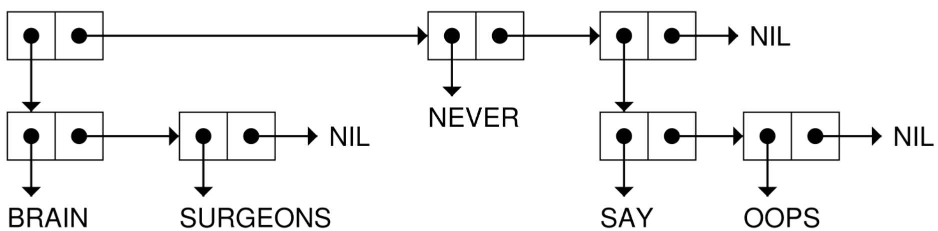
- Each of the  boxes in such a drawing represents a Lisp data object called a ***cons cell***.
- A cons cell has 2 fields that contain pointers. (Pointers are represented by the arrows in box and arrow drawings.)
  - The pointer in one field is called the ***car*** pointer; this is shown on the ***left*** side of a cons cell box.
  -

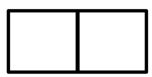


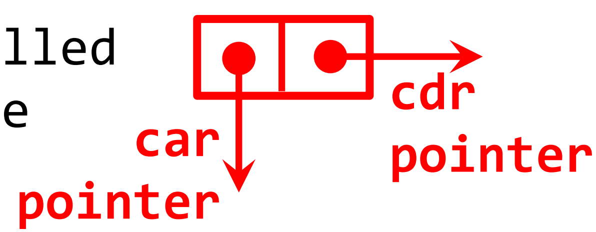
We have already seen "*box* and *arrow*" drawings of Lisp data objects represented by lists. Below is one example:

From p. 34 of Touretzky.

((BRAIN SURGEONS) NEVER (SAY OOPS)) represents:



- Each of the  boxes in such a drawing represents a Lisp data object called a ***cons cell***.
- A cons cell has 2 fields that contain pointers. (Pointers are represented by the arrows in box and arrow drawings.)
  - The pointer in one field is called the ***car*** pointer; this is shown on the ***left*** side of a cons cell box.
  - The pointer in the other field is called the ***cdr*** pointer; this is shown on the ***right*** side of a cons cell box.



Lisp implementations deal with S-expressions via pointers:

-

Lisp implementations deal with S-expressions via pointers:

- When passing an S-expression as argument to a function call or returning an S-expression as the result of a function call, *what do we actually pass or return*?

  Answer:

Lisp implementations deal with S-expressions via pointers:

- When passing an S-expression as argument to a function call or returning an S-expression as the result of a function call, *what do we actually pass or return*?

  Answer: We pass or return a ***pointer*** *to the S-expression's data object*.

  -

Lisp implementations deal with S-expressions via pointers:

- When passing an S-expression as argument to a function call or returning an S-expression as the result of a function call, *what do we actually pass or return*?

  Answer: We pass or return a ***pointer*** *to the S-expression's data object*.

  - In some implementations this may not actually be true when the S-expression in question is a number or a character, but

Lisp implementations deal with S-expressions via pointers:

- When passing an S-expression as argument to a function call or returning an S-expression as the result of a function call, *what do we actually pass or return*?

  Answer: We pass or return a ***pointer*** *to the S-expression's data object*.

  - In some implementations this may not actually be true when the S-expression in question is a number or a character, but even then it's OK for Lisp programmers to assume it's true, as that assumption won't lead to wrong predictions about the *observable behavior* of the code.

Lisp implementations deal with S-expressions via pointers:

- When passing an S-expression as argument to a function call or returning an S-expression as the result of a function call, *what do we actually pass or return*?

  Answer: We pass or return a ***pointer*** *to the S-expression's data object*.

Lisp implementations deal with S-expressions via pointers:

- When passing an S-expression as argument to a function call or returning an S-expression as the result of a function call, *what do we actually pass or return*?

  Answer: We pass or return a ***pointer** to the S-expression's data object*.

- Every nonempty list has a "root" cons cell, which is *the unique cons cell of the list that is **not** pointed at by a car or cdr pointer of another cons cell of the list.*
  Example:

Lisp implementations deal with S-expressions via pointers:

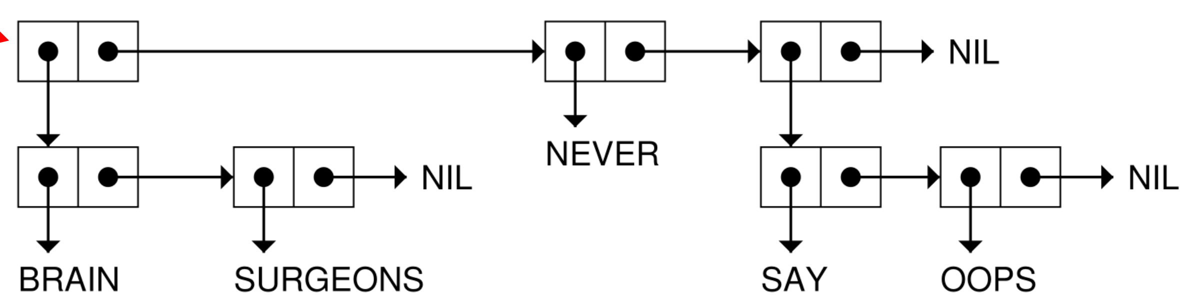
- When passing an S-expression as argument to a function call or returning an S-expression as the result of a function call, *what do we actually pass or return*?

  Answer: We pass or return a ***pointer** to the S-expression's data object*.

- Every nonempty list has a "root" cons cell, which is *the unique cons cell of the list that is **not** pointed at by a car or cdr pointer of another cons cell of the list.*

  Example:

  This cons cell is the "root" cons cell of the list.



-

Lisp implementations deal with S-expressions via pointers:

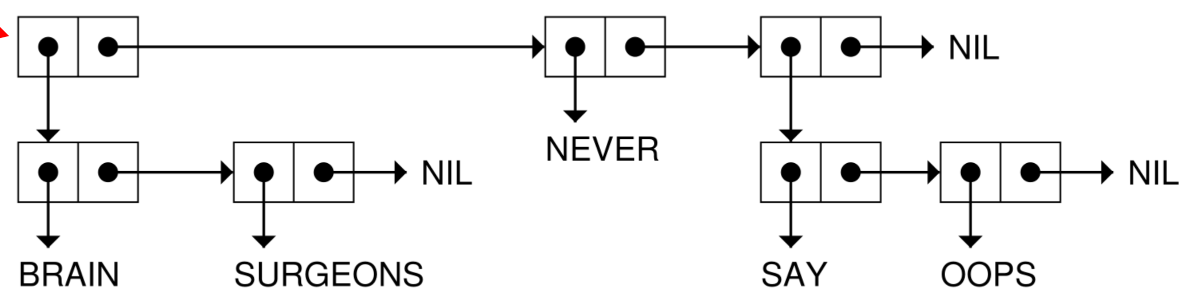
- When passing an S-expression as argument to a function call or returning an S-expression as the result of a function call, *what do we actually pass or return*?

  Answer: We pass or return a ***pointer to the S-expression's data object***.

- Every nonempty list has a "root" cons cell, which is *the unique cons cell of the list that is **not** pointed at by a car or cdr pointer of another cons cell of the list.*

  Example:

  This cons cell is the "root" cons cell of the list.



- When passing a nonempty list as argument to a function call, or returning a nonempty list as the result of a function call, we pass or return ***a pointer to the "root" cons cell of that list***.

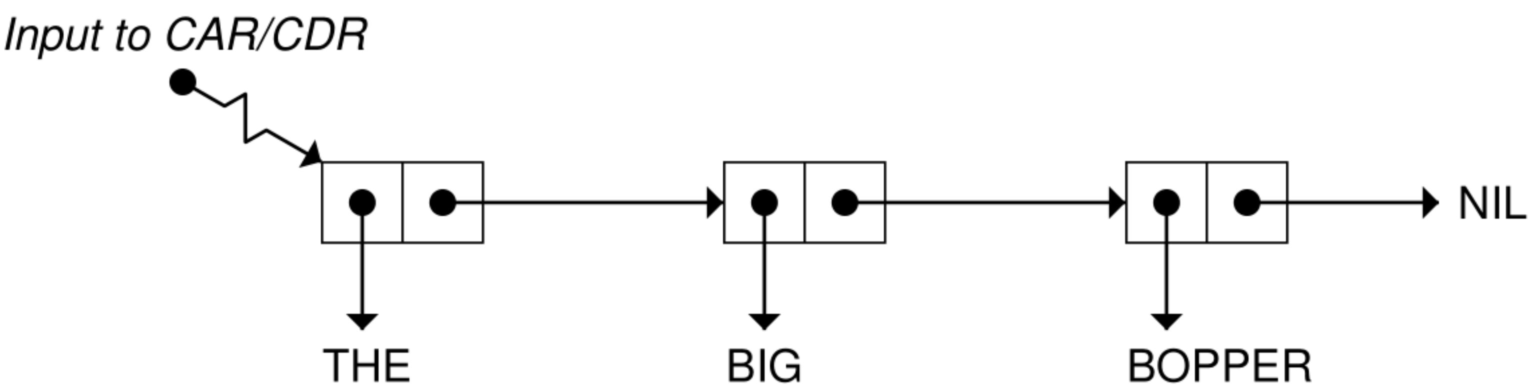
```
p. 43 of Touretzky explains the evaluation of
           (CAR '(THE BIG BOPPER))
as follows:
```

p. 43 of Touretzky explains the evaluation of

```
(CAR '(THE BIG BOPPER))
```

as follows:

When this list is used as input to a function such as CAR, what the function actually receives is not the list itself, but rather a pointer to the first cons cell:



CAR follows this pointer to get to the actual cons cell and extracts the pointer sitting in the CAR half. So CAR returns as its result a pointer to the symbol THE.

p. 43 of Touretzky then explains the evaluation of

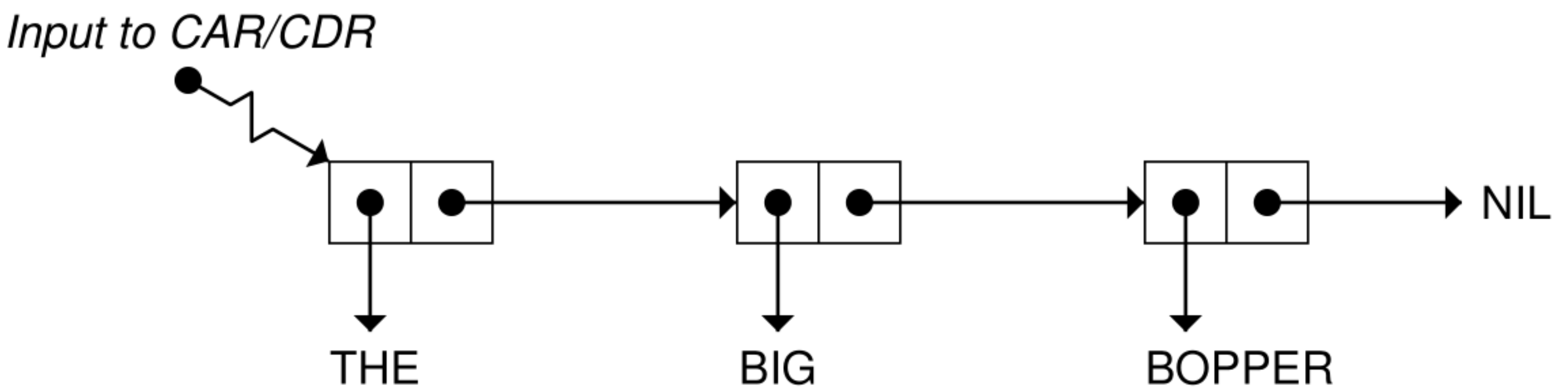
```
(CDR '(THE BIG BOPPER))
```

as follows:

p. 43 of Touretzky then explains the evaluation of

(CDR '(THE BIG BOPPER))

as follows:



CDR follows the pointer to get to the cons cell, and extracts the pointer sitting in the CDR half, which it returns. So the result of CDR is a pointer to the list (BIG BOPPER).

A function call (CAR *L*) is evaluated as follows:

1.

2.

3.

4.

A function call (CAR *l*) is evaluated as follows:

1. Evaluate *l*: This yields a pointer to the data object that is *l*'s value.
2.

3.
4.

A function call (CAR *l*) is evaluated as follows:

1. Evaluate *l*: This yields a pointer to the data object that is *l*'s value.
2. If *l*'s value is a ***nonempty list***, then the pointer given by step 1 points to the list's "root" cons cell.

   In this case we **return** $\boxed{}$
3.
4.

A function call (CAR *l*) is evaluated as follows:

1. Evaluate *l*: This yields a pointer to the data object that is *l*'s value.
2. If *l*'s value is a ***nonempty list***, then the pointer given by step 1 points to the list's "root" cons cell.

   In this case we **return** that cons cell's ***car*** pointer.
3.
4.

A function call (CAR *l*) is evaluated as follows:

1. Evaluate *l*: This yields a pointer to the data object that is *l*'s value.
2. If *l*'s value is a ***nonempty list***, then the pointer given by step 1 points to the list's "root" cons cell.
   In this case we **return** that cons cell's ***car*** pointer.
3. If *l*'s value is **NIL**, then we **return** a pointer to .
4.

A function call (CAR *l*) is evaluated as follows:

1. Evaluate *l*: This yields a pointer to the data object that is *l*'s value.
2. If *l*'s value is a ***nonempty list***, then the pointer given by step 1 points to the list's "root" cons cell.
   In this case we **return** that cons cell's ***car*** pointer.
3. If *l*'s value is **NIL**, then we **return** a pointer to **NIL**.
4.

A function call (CAR *l*) is evaluated as follows:

1. Evaluate *l*: This yields a pointer to the data object that is *l*'s value.
2. If *l*'s value is a ***nonempty list***, then the pointer given by step 1 points to the list's "root" cons cell.

   In this case we **return** that cons cell's ***car*** pointer.
3. If *l*'s value is **NIL**, then we **return** a pointer to **NIL**.
4. Otherwise *l*'s value isn't a list, so we ***report an error***.

A function call (CAR *l*) is evaluated as follows:

1. Evaluate *l*: This yields a pointer to the data object that is *l*'s value.
2. If *l*'s value is a ***nonempty list***, then the pointer given by step 1 points to the list's "root" cons cell.

   In this case we **return** that cons cell's ***car*** pointer.
3. If *l*'s value is **NIL**, then we **return** a pointer to **NIL**.
4. Otherwise *l*'s value isn't a list, so we ***report an error***.

A function call (CDR *l*) is evaluated in the same way, but with *cdr* instead of *car* at step 2:

A function call (CAR *l*) is evaluated as follows:

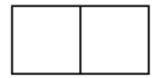
1. Evaluate *l*: This yields a pointer to the data object that is *l*'s value.
2. If *l*'s value is a ***nonempty list***, then the pointer given by step 1 points to the list's "root" cons cell.
   In this case we **return** that cons cell's ***car*** pointer.
3. If *l*'s value is **NIL**, then we **return** a pointer to **NIL**.
4. Otherwise *l*'s value isn't a list, so we ***report an error***.

A function call (CDR *l*) is evaluated in the same way, but with *cdr* instead of *car* at step 2:

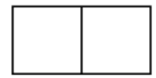
1. Evaluate *l*: This yields a pointer to the data object that is *l*'s value.
2. If *l*'s value is a ***nonempty list***, then the pointer given by step 1 points to the list's "root" cons cell.
   In this case we **return** that cons cell's ***cdr*** pointer.
3. If *l*'s value is **NIL**, then we **return** a pointer to **NIL**.
4. Otherwise *l*'s value isn't a list, so we ***report an error***.

p. 54 of Touretzky explains the evaluation of (CONS 'HELLO '(THERE MISS DOOLITTLE)) as follows:

p. 54 of Touretzky explains the evaluation of (CONS 'HELLO '(THERE MISS DOOLITTLE)) as follows:

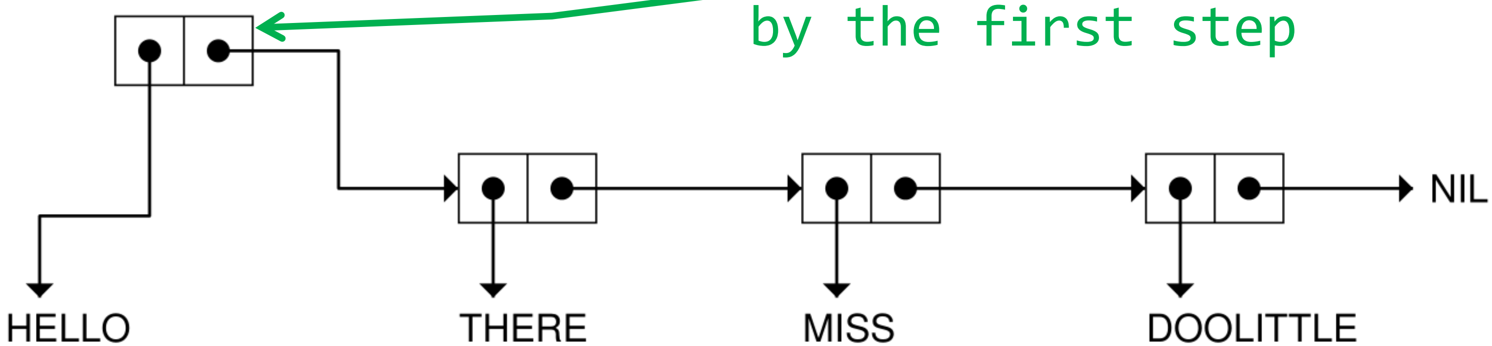
CONS creates a new cons cell: 

p. 54 of Touretzky explains the evaluation of (CONS 'HELLO '(THERE MISS DOOLITTLE)) as follows:

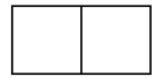
CONS creates a new cons cell: 

It fills in the CAR and CDR pointers:

The new cons cell created by the first step

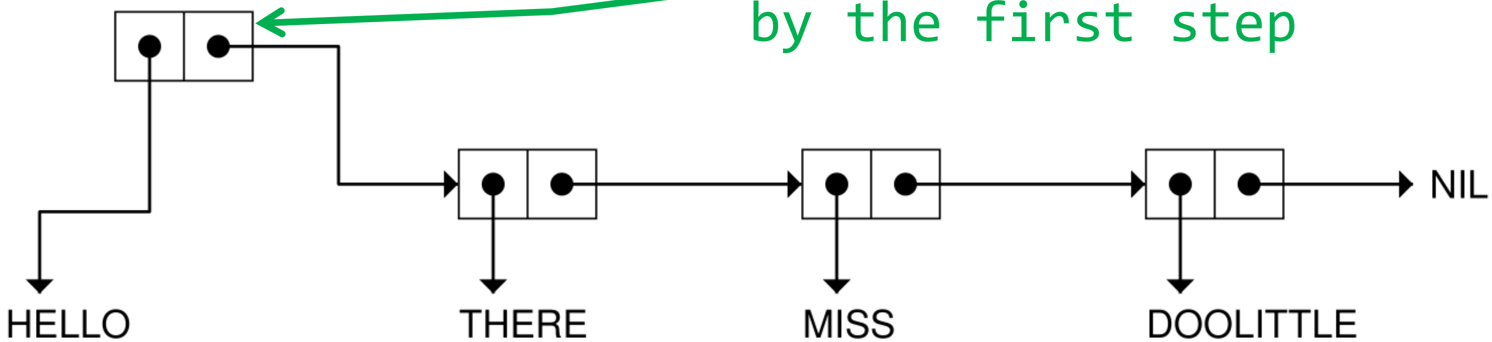


p. 54 of Touretzky explains the evaluation of (CONS 'HELLO '(THERE MISS DOOLITTLE)) as follows:

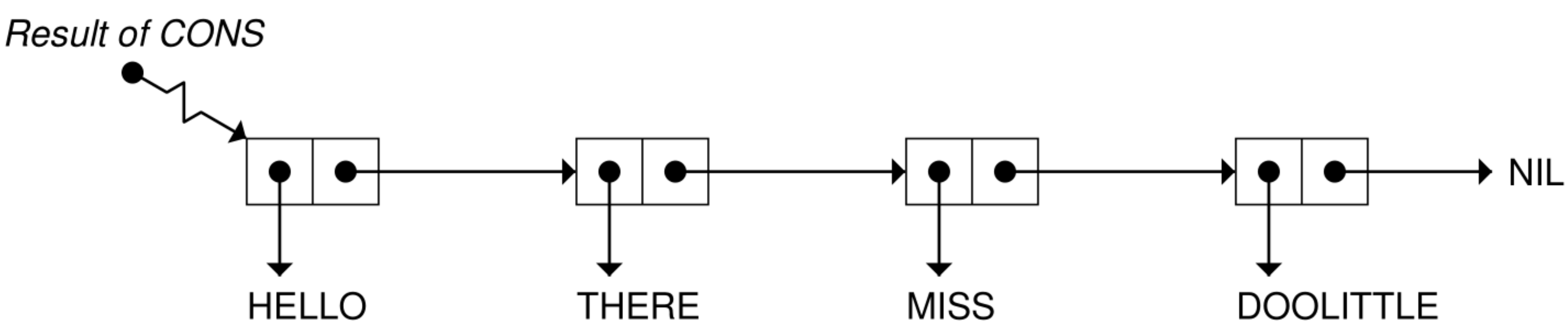
CONS creates a new cons cell: 

It fills in the CAR and CDR pointers:

The new cons cell created by the first step



And it returns a pointer to the new cell, which is now the head of a cons cell chain one longer than CONS's second input:



A function call (CONS *x* *L*) is evaluated as follows:

1.

2.

3.

4.

5.

A function call (CONS *x* *l*) is evaluated as follows:

1. Evaluate *x* and *l*: This yields pointers to 2 data objects, namely *x*'s value and *l*'s value.
2.
3.
4.
5.

A function call (CONS *x* *l*) is evaluated as follows:

1. Evaluate *x* and *l*: This yields pointers to 2 data objects, namely *x*'s value and *l*'s value.

2. Create a new ***cons*** cell.

3.

4.

5.

A function call (CONS *x* *l*) is evaluated as follows:

1. Evaluate *x* and *l*: This yields pointers to 2 data objects, namely *x*'s value and *l*'s value.
2. Create a new ***cons*** cell.
3. Store the pointer to *x*'s value that was found at step 1 in the ***car*** pointer field of the new cons cell.
4.
5.

A function call (CONS *x* *l*) is evaluated as follows:

1. Evaluate *x* and *l*: This yields pointers to 2 data objects, namely *x*'s value and *l*'s value.
2. Create a new ***cons*** cell.
3. Store the pointer to *x*'s value that was found at step 1 in the ***car*** pointer field of the new cons cell.
4. Store the pointer to *l*'s value that was found at step 1 in the ***cdr*** pointer field of the new cons cell.
5.

A function call (CONS *x* *l*) is evaluated as follows:

1. Evaluate *x* and *l*: This yields pointers to 2 data objects, namely *x*'s value and *l*'s value.

2. Create a new ***cons*** cell.

3. Store the pointer to *x*'s value that was found at step 1 in the ***car*** pointer field of the new cons cell.

4. Store the pointer to *l*'s value that was found at step 1 in the ***cdr*** pointer field of the new cons cell.

5. Return a pointer to the new cons cell.

A function call (CONS *x* *l*) is evaluated as follows:

1. Evaluate *x* and *l*: This yields pointers to 2 data objects, namely *x*'s value and *l*'s value.
2. Create a new ***cons*** cell.
3. Store the pointer to *x*'s value that was found at step 1 in the ***car*** pointer field of the new cons cell.
4. Store the pointer to *l*'s value that was found at step 1 in the ***cdr*** pointer field of the new cons cell.
5. Return a pointer to the new cons cell.

- See p. 59 of Touretzky for an explanation of how a function call **(LIST 'FOO 'BAR 'BAZ)** is evaluated.
-

A function call (CONS *x* *l*) is evaluated as follows:

1. Evaluate *x* and *l*: This yields pointers to 2 data objects, namely *x*'s value and *l*'s value.
2. Create a new ***cons*** cell.
3. Store the pointer to *x*'s value that was found at step 1 in the ***car*** pointer field of the new cons cell.
4. Store the pointer to *l*'s value that was found at step 1 in the ***cdr*** pointer field of the new cons cell.
5. Return a pointer to the new cons cell.

- See p. 59 of Touretzky for an explanation of how a function call **(LIST 'FOO 'BAR 'BAZ)** is evaluated.

- See pp. 162 – 3 of Touretzky for an explanation of how a function call **(APPEND '(A B C) '(D E))** is evaluated.

Predicates and Conditionals

- In Lisp, a ***predicate*** is a function whose calls return values that represent ***true*** or ***false***.
-
-

- In Lisp, a ***predicate*** is a function whose calls return values that represent ***true*** or ***false***.
- In Common Lisp:
  - ***False*** is represented by the symbol NIL.
  - ***True*** can be represented by
    -
  -
  -
-

- In Lisp, a ***predicate*** is a function whose calls return values that represent ***true*** or ***false***.
- In Common Lisp:
  - ***False*** is represented by the symbol NIL.
  - ***True*** can be represented by ***any value other than*** NIL:
    - **T**, **19.5**, **0**, **""**, **DOG**, and **(A (B C) D)** all represent *true*!
  -
  -
-

- In Lisp, a ***predicate*** is a function whose calls return values that represent ***true*** or ***false***.
- In Common Lisp:
  - ***False*** is represented by the symbol NIL.
  - ***True*** can be represented by ***any value other than*** NIL:
    - **T**, **19.5**, **0**, **""**, **DOG**, and **(A (B C) D)** all represent *true*!
  - The symbol T is the *usual* way to represent ***true***: Use some other value only if there's a good reason!

  Exercise:

  Answer:

  Exercise:

  Answer:

  -

-

- In Lisp, a ***predicate*** is a function whose calls return values that represent ***true*** or ***false***.
- In Common Lisp:
  - ***False*** is represented by the symbol NIL.
  - ***True*** can be represented by ***any value other than*** NIL:
    - **T**, **19.5**, **0**, **""**, **DOG**, and **(A (B C) D)** all represent *true*!
  - The symbol T is the *usual* way to represent ***true***: Use some other value only if there's a good reason!

  Exercise: What is the value of **(if 0 1 2)** in Lisp?

  Answer:

  Exercise: What is the value of **(if () 1 2)** in Lisp?

  Answer:

  -

-

- In Lisp, a ***predicate*** is a function whose calls return values that represent ***true*** or ***false***.
- In Common Lisp:
  - ***False*** is represented by the symbol NIL.
  - ***True*** can be represented by ***any value other than*** NIL:
    - **T**, **19.5**, **0**, **""**, **DOG**, and **(A (B C) D)** all represent *true*!
  - The symbol T is the *usual* way to represent ***true***: Use some other value only if there's a good reason!

  Exercise: What is the value of **(if 0 1 2)** in Lisp?

  Answer: As 0 represents *true*, the value is 1.

  Exercise: What is the value of **(if () 1 2)** in Lisp?

  Answer: As () is the same as NIL, the value is 2.

- In Lisp, a ***predicate*** is a function whose calls return values that represent ***true*** or ***false***.
- In Common Lisp:
  - ***False*** is represented by the symbol NIL.
  - ***True*** can be represented by ***<u>any value other than</u>*** NIL:
    - **T**, **19.5**, **0**, **""**, **DOG**, and **(A (B C) D)** all represent *true*!
  - The symbol T is the *<u>usual</u>* way to represent ***true***: Use some other value only if there's a good reason!

  Exercise: What is the value of **(if 0 1 2)** in Lisp?

  Answer: As 0 represents *true*, the value is 1.

  Exercise: What is the value of **(if () 1 2)** in Lisp?

  Answer: As () is the same as NIL, the value is 2.

  - The fact that () represents false will actually be important when we consider the predicate MEMBER!
-

- In Lisp, a ***predicate*** is a function whose calls return values that represent ***true*** or ***false***.
- In Common Lisp:
  - ***False*** is represented by the symbol NIL.
  - ***True*** can be represented by ***<u>any value other than</u>*** NIL:
    - **T**, **19.5**, **0**, **""**, **DOG**, and **(A (B C) D)** all represent *true*!
  - The symbol T is the *<u>usual</u>* way to represent ***true***: Use some other value only if there's a good reason!

  Exercise: What is the value of **(if 0 1 2)** in Lisp?

  Answer: As 0 represents *true*, the value is 1.

  Exercise: What is the value of **(if () 1 2)** in Lisp?

  Answer: As () is the same as NIL, the value is 2.
  - The fact that () represents false will actually be important when we consider the predicate MEMBER!
- Recall that T and NIL are constant symbols that evaluate to themselves: So T and NIL never have to be quoted, just as numbers never have to be quoted!

Equality Predicates

Equality Predicates

- Lisp has several predicates for testing equality, of which the following four are the most commonly used:
  -
  -
  -
  -
-

Equality Predicates

- Lisp has several predicates for testing equality, of which the following four are the most commonly used:
  - equal
  - eql
  - eq
  - =
-

Equality Predicates

- Lisp has several predicates for testing equality, of which the following four are the most commonly used:
  - equal
  - eql
  - eq
  - =
- (equal *x* *y*) ⇒ T if the argument values are equal
  (equal *x* *y*) ⇒ NIL if the argument values are not equal

Equality Predicates

- Lisp has several predicates for testing equality, of which the following four are the most commonly used:
  - `equal`
  - `eql`
  - `eq`
  - `=`

- (equal *x* *y*) ⇒ T if the argument values are equal
  (equal *x* *y*) ⇒ NIL if the argument values are not equal
  - (equal (car '(a b c)) (cadr '(1 a b c))) ⇒
  - (equal (cdr '(a b c)) (cdr '(1 a b c))) ⇒
  - (equal (list 'a 'b 'c) (cdr '(1 a b c))) ⇒
  - (equal (+ 1 2) 3) ⇒
  - (equal (+ 1 2) 3.0) ⇒
  - (equal (+ 1.0 2) 3.0) ⇒
  - (equal 0.5 1/2) ⇒
  - (equal (/ 1 2) 1/2) ⇒
  - (equal 0.5 (/ 1 2)) ⇒

Equality Predicates

- Lisp has several predicates for testing equality, of which the following four are the most commonly used:
  - `equal`
  - `eql`
  - `eq`
  - `=`
- `(equal` *x y*`) ⇒ T  if the argument values are equal`
  `(equal` *x y*`) ⇒ NIL if the argument values are not equal`
  - `(equal (car '(a b c)) (cadr '(1 a b c))) ⇒ T`
  - `(equal (cdr '(a b c)) (cdr '(1 a b c))) ⇒`
  - `(equal (list 'a 'b 'c) (cdr '(1 a b c))) ⇒`
  - `(equal (+ 1 2) 3) ⇒`
  - `(equal (+ 1 2) 3.0) ⇒`
  - `(equal (+ 1.0 2) 3.0) ⇒`
  - `(equal 0.5 1/2) ⇒`
  - `(equal (/ 1 2) 1/2) ⇒`
  - `(equal 0.5 (/ 1 2)) ⇒`

Equality Predicates

- Lisp has several predicates for testing equality, of which the following four are the most commonly used:
  - equal
  - eql
  - eq
  - =

- (equal *x y*) ⇒ T if the argument values are equal
  (equal *x y*) ⇒ NIL if the argument values are not equal
  - (equal (car '(a b c)) (cadr '(1 a b c))) ⇒ T
  - (equal (cdr '(a b c)) (cdr '(1 a b c))) ⇒ NIL
  - (equal (list 'a 'b 'c) (cdr '(1 a b c))) ⇒
  - (equal (+ 1 2) 3) ⇒
  - (equal (+ 1 2) 3.0) ⇒
  - (equal (+ 1.0 2) 3.0) ⇒
  - (equal 0.5 1/2) ⇒
  - (equal (/ 1 2) 1/2) ⇒
  - (equal 0.5 (/ 1 2)) ⇒

Equality Predicates

- Lisp has several predicates for testing equality, of which the following four are the most commonly used:
  - equal
  - eql
  - eq
  - =

- (equal *x* *y*) ⇒ T if the argument values are equal
  (equal *x* *y*) ⇒ NIL if the argument values are not equal
  - (equal (car '(a b c)) (cadr '(1 a b c))) ⇒ T
  - (equal (cdr '(a b c)) (cdr '(1 a b c))) ⇒ NIL
  - (equal (list 'a 'b 'c) (cdr '(1 a b c))) ⇒ T
  - (equal (+ 1 2) 3) ⇒
  - (equal (+ 1 2) 3.0) ⇒
  - (equal (+ 1.0 2) 3.0) ⇒
  - (equal 0.5 1/2) ⇒
  - (equal (/ 1 2) 1/2) ⇒
  - (equal 0.5 (/ 1 2)) ⇒

Equality Predicates

- Lisp has several predicates for testing equality, of which the following four are the most commonly used:
  - equal
  - eql
  - eq
  - =
- (equal *x* *y*) ⇒ T if the argument values are equal
  (equal *x* *y*) ⇒ NIL if the argument values are not equal
  - (equal (car '(a b c)) (cadr '(1 a b c))) ⇒ T
  - (equal (cdr '(a b c)) (cdr '(1 a b c))) ⇒ NIL
  - (equal (list 'a 'b 'c) (cdr '(1 a b c))) ⇒ T
  - (equal (+ 1 2) 3) ⇒ T
  - (equal (+ 1 2) 3.0) ⇒
  - (equal (+ 1.0 2) 3.0) ⇒
  - (equal 0.5 1/2) ⇒
  - (equal (/ 1 2) 1/2) ⇒
  - (equal 0.5 (/ 1 2)) ⇒

Equality Predicates

- Lisp has several predicates for testing equality, of which the following four are the most commonly used:
  - equal
  - eql
  - eq
  - =

- (equal *x y*) ⇒ T if the argument values are equal
  (equal *x y*) ⇒ NIL if the argument values are not equal
  - (equal (car '(a b c)) (cadr '(1 a b c))) ⇒ T
  - (equal (cdr '(a b c)) (cdr '(1 a b c))) ⇒ NIL
  - (equal (list 'a 'b 'c) (cdr '(1 a b c))) ⇒ T
  - (equal (+ 1 2) 3) ⇒ T
  - (equal (+ 1 2) 3.0) ⇒ NIL
  - (equal (+ 1.0 2) 3.0) ⇒
  - (equal 0.5 1/2) ⇒
  - (equal (/ 1 2) 1/2) ⇒
  - (equal 0.5 (/ 1 2)) ⇒

Equality Predicates

- Lisp has several predicates for testing equality, of which the following four are the most commonly used:
  - `equal`
  - `eql`
  - `eq`
  - `=`
- (equal *x* *y*) ⇒ T if the argument values are equal
  (equal *x* *y*) ⇒ NIL if the argument values are not equal
  - (equal (car '(a b c)) (cadr '(1 a b c))) ⇒ T
  - (equal (cdr '(a b c)) (cdr '(1 a b c))) ⇒ NIL
  - (equal (list 'a 'b 'c) (cdr '(1 a b c))) ⇒ T
  - (equal (+ 1 2) 3) ⇒ T
  - (equal (+ 1 2) 3.0) ⇒ NIL
  - (equal (+ 1.0 2) 3.0) ⇒ T
  - (equal 0.5 1/2) ⇒
  - (equal (/ 1 2) 1/2) ⇒
  - (equal 0.5 (/ 1 2)) ⇒

Equality Predicates

- Lisp has several predicates for testing equality, of which the following four are the most commonly used:
  - equal
  - eql
  - eq
  - =

- (equal *x* *y*) ⇒ T if the argument values are equal
  (equal *x* *y*) ⇒ NIL if the argument values are not equal
  - (equal (car '(a b c)) (cadr '(1 a b c))) ⇒ T
  - (equal (cdr '(a b c)) (cdr '(1 a b c))) ⇒ NIL
  - (equal (list 'a 'b 'c) (cdr '(1 a b c))) ⇒ T
  - (equal (+ 1 2) 3) ⇒ T
  - (equal (+ 1 2) 3.0) ⇒ NIL
  - (equal (+ 1.0 2) 3.0) ⇒ T
  - (equal 0.5 1/2) ⇒ NIL
  - (equal (/ 1 2) 1/2) ⇒
  - (equal 0.5 (/ 1 2)) ⇒

Equality Predicates

- Lisp has several predicates for testing equality, of which the following four are the most commonly used:
    - equal
    - eql
    - eq
    - =
- (equal *x* *y*) ⇒ T if the argument values are equal
  (equal *x* *y*) ⇒ NIL if the argument values are not equal
    - (equal (car '(a b c)) (cadr '(1 a b c))) ⇒ T
    - (equal (cdr '(a b c)) (cdr '(1 a b c))) ⇒ NIL
    - (equal (list 'a 'b 'c) (cdr '(1 a b c))) ⇒ T
    - (equal (+ 1 2) 3) ⇒ T
    - (equal (+ 1 2) 3.0) ⇒ NIL
    - (equal (+ 1.0 2) 3.0) ⇒ T
    - (equal 0.5 1/2) ⇒ NIL
    - (equal (/ 1 2) 1/2) ⇒ T
    - (equal 0.5 (/ 1 2)) ⇒

Equality Predicates

- Lisp has several predicates for testing equality, of which the following four are the most commonly used:
  - equal
  - eql
  - eq
  - =

- (equal *x* *y*) ⇒ T if the argument values are equal
  (equal *x* *y*) ⇒ NIL if the argument values are not equal
  - (equal (car '(a b c)) (cadr '(1 a b c))) ⇒ T
  - (equal (cdr '(a b c)) (cdr '(1 a b c))) ⇒ NIL
  - (equal (list 'a 'b 'c) (cdr '(1 a b c))) ⇒ T
  - (equal (+ 1 2) 3) ⇒ T
  - (equal (+ 1 2) 3.0) ⇒ NIL
  - (equal (+ 1.0 2) 3.0) ⇒ T
  - (equal 0.5 1/2) ⇒ NIL
  - (equal (/ 1 2) 1/2) ⇒ T
  - (equal 0.5 (/ 1 2)) ⇒ NIL