Lists

- If a is an atom, then $(e_1 \ \ldots \ e_n \ . \ a)$ represents a singly-linked list data structure that may be drawn as follows:

- A *proper* list $(e_1 \ \ldots \ e_n)$ represents a singly-linked list data structure that may be drawn as follows:

- If a is a list, then $(e_1 \ \ldots \ e_n \ . \ a)$ can be simplified:

$(e_1 \ \ldots \ e_n \ . \ \text{NIL}) = (e_1 \ \ldots \ e_n)$

$(e_1 \ \ldots \ e_n \ . \ (e_{n+1})) = (e_1 \ \ldots \ e_n \ e_{n+1})$

$(e_1 \ \ldots \ e_n \ . \ (e_{n+1} \ \ldots \ e_{n+k})) = (e_1 \ \ldots \ e_n \ e_{n+1} \ \ldots \ e_{n+k})$

$(e_1 \ \ldots \ e_n \ . \ (e_{n+1} \ \ldots \ e_{n+k} \ . \ a)) = (e_1 \ \ldots \ e_n \ e_{n+1} \ \ldots \ e_{n+k} \ . \ a)$

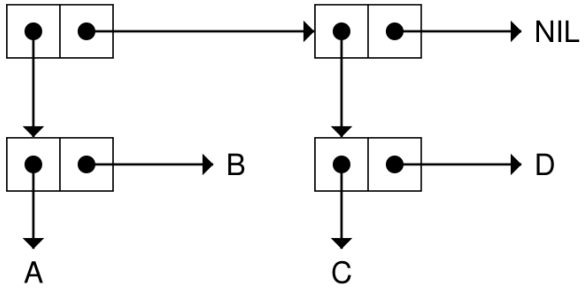
- So we'll write $(e_1 \ldots e_n \ . \ a)$ ***only when a is an atom other than*** NIL.

Lists

- If a is an atom, then $(e_1 \ \ldots \ e_n \ . \ a)$ represents a singly-linked list data structure that may be drawn as follows:

- A *proper* list $(e_1 \ \ldots \ e_n)$ represents a singly-linked list data structure that may be drawn as follows:

((A . B) (C . D)) is a *proper list of 2 dotted lists* that represents:



This example is from p. 73 and p. C-18 of Touretzky.

Further Comments on Dotted Lists

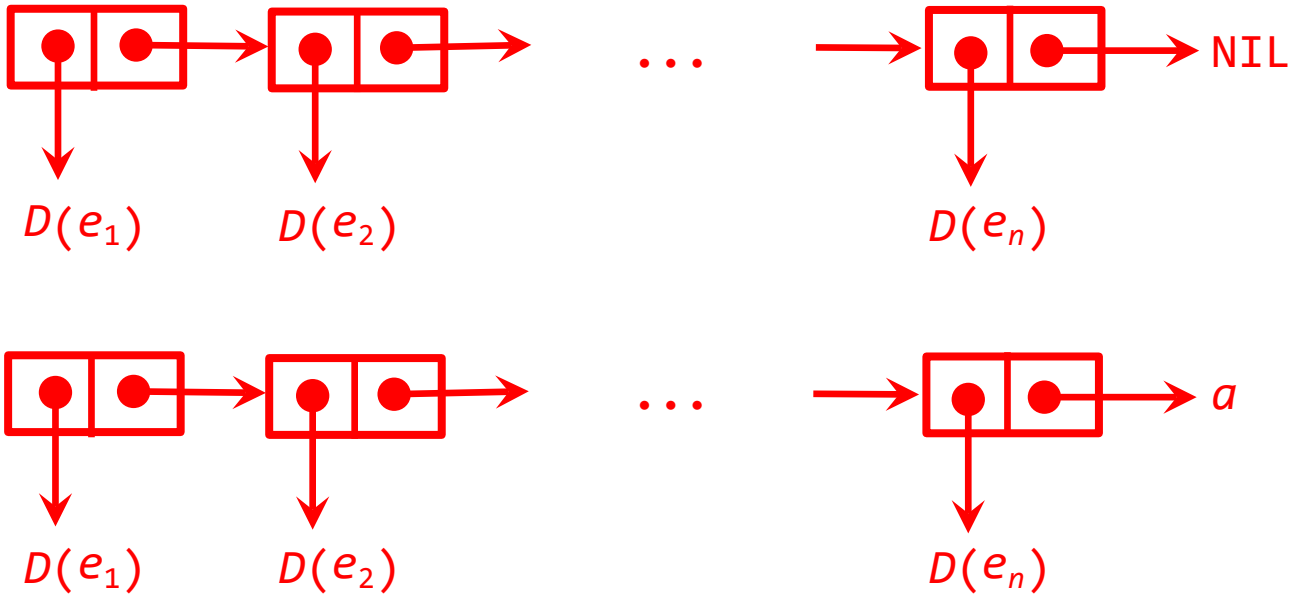
- If a is an atom, then $(e_1 \ \ldots \ e_n \ . \ a)$ represents a singly-linked list data structure that may be drawn as follows:

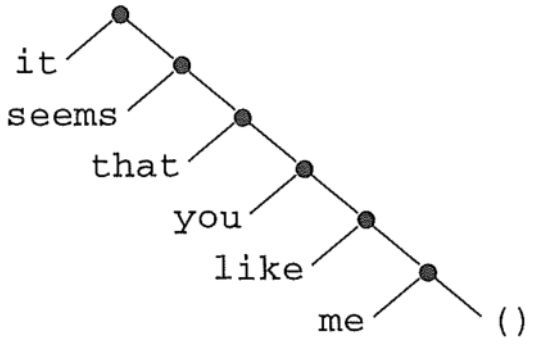
- A *proper* list $(e_1 \ \ldots \ e_n)$ represents a singly-linked list data structure that may be drawn as follows:

- If a is an atom, the ***length*** of $(e_1 \ldots e_n \ . \ a)$ is n (***not*** n+1).
- Dotted lists are used much less than proper lists.
- The term ***list*** is often used to mean ***proper list***!
- If a function you write for this course returns a dotted list, then either your code has a bug or an inappropriate argument value was passed to the function!

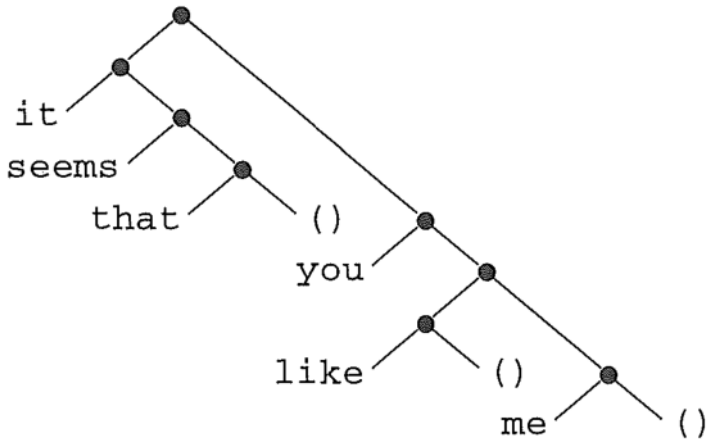
Tree Representation of S-Expressions

These "box and arrow" drawings are quite often simplified to ***binary trees*** in which the two children of any internal node v are ***either*** south and east of v ***or*** southwest and southeast of v:





(it seems that you like me)

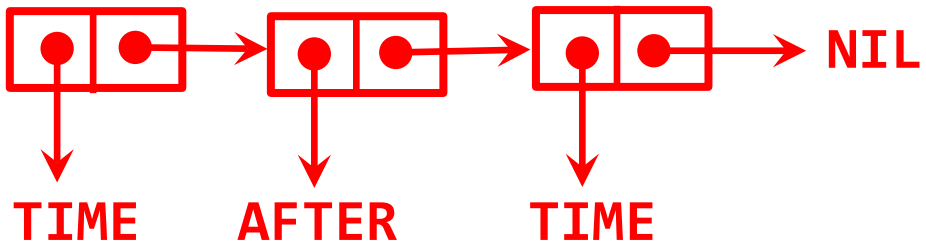


((it seems that) you (like) me)

These two examples are from p. 393 of Sethi's book.

Memory Uniqueness of Symbols

(TIME AFTER TIME) represents:



But this drawing needs careful interpretation because the two occurrences of TIME represent ***the same identical*** ***symbol object***.

Touretzky explains this as follows on p. 195:

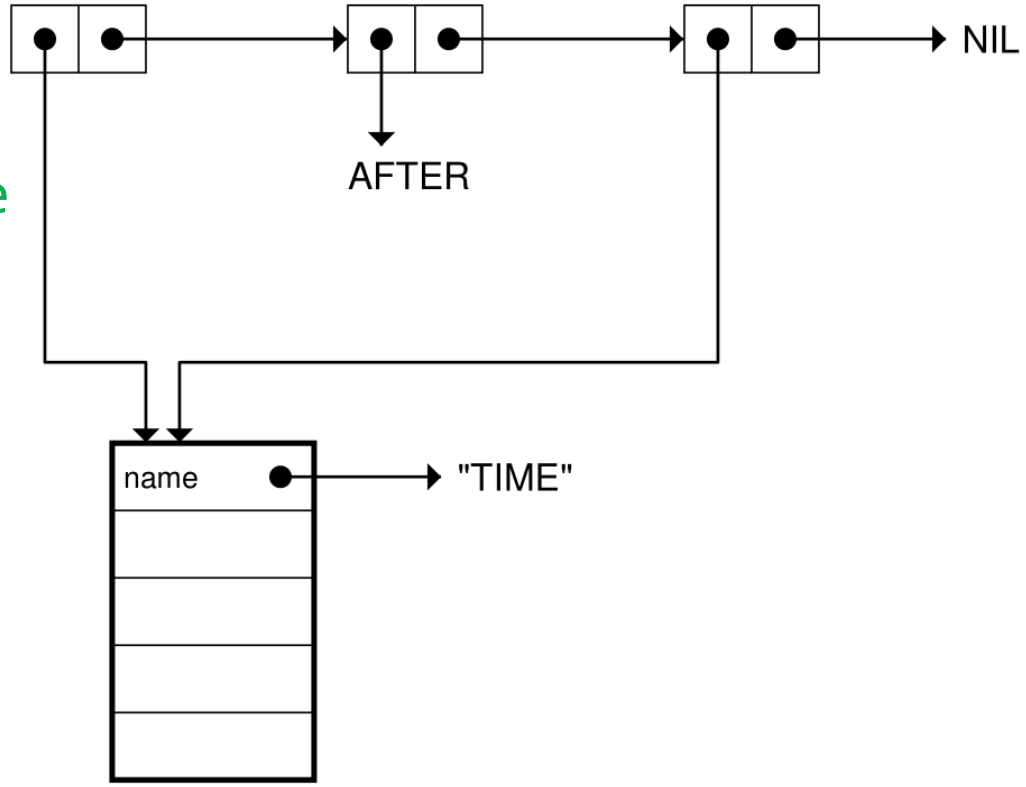
In Lisp, symbols are unique, meaning there can be only one symbol in the computer's memory with a given name.** Every object in the memory has a numbered location, called its **address**. Since a symbol exists in only one place in memory, symbols have unique addresses. So in the list (TIME AFTER TIME), the two occurrences of the symbol TIME must refer to the same address. There cannot be two separate symbols named TIME.

Memory Uniqueness of Symbols

In Lisp, symbols are unique, meaning there can be only one symbol in the computer's memory with a given name.** Every object in the memory has a numbered location, called its **address**. Since a symbol exists in only one place in memory, symbols have unique addresses. So in the list (TIME AFTER TIME), the two occurrences of the symbol TIME must refer to the same address. There cannot be two separate symbols named TIME.

The following more detailed depiction of the data structure represented by **(TIME AFTER TIME)** is given on p. 196 of Touretzky:

There is ***just one*** TIME ***symbol object***!



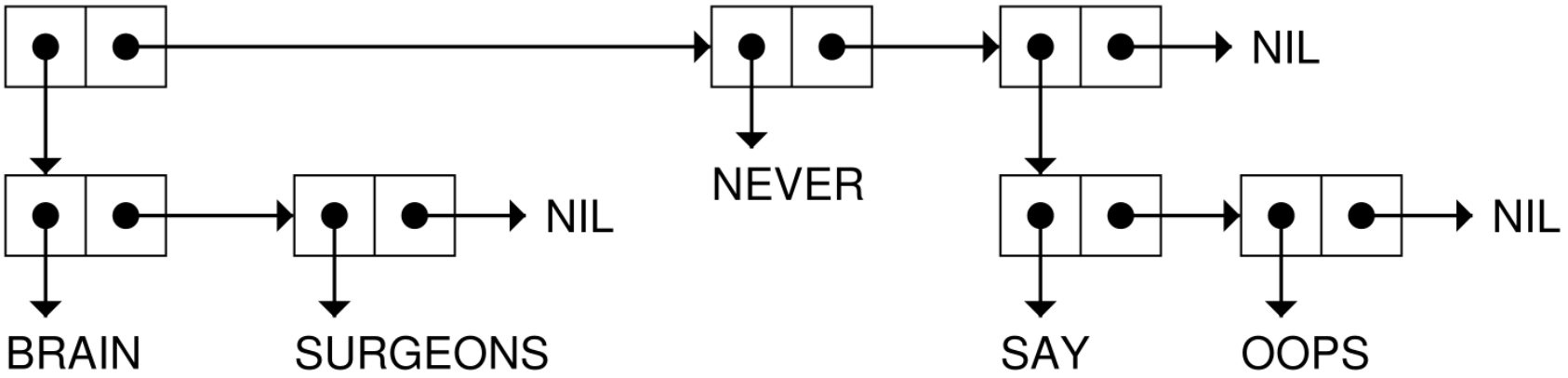
Memory Uniqueness of Symbols

In Lisp, symbols are unique, meaning there can be only one symbol in the computer's memory with a given name.** Every object in the memory has a numbered location, called its **address**. Since a symbol exists in only one place in memory, symbols have unique addresses. So in the list (TIME AFTER TIME), the two occurrences of the symbol TIME must refer to the same address. There cannot be two separate symbols named TIME.

Similarly, all occurrences of the symbol NIL that are shown below represent ***the same identical symbol object***:

From p. 34 of Touretzky.

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



QUOTE is a special operator you will use most often!

(QUOTE *e*) evaluates to *e*; *e* is not evaluated!

E.g.: The value of **(QUOTE (+ 3 4))** is the list **(+ 3 4)**.

Note: (QUOTE e_1 … e_n) cannot be evaluated if $n \neq 1$; evaluation produces an error!

For any S-expression *e*, **'*e* is equivalent to (QUOTE *e*)** and we usually write 'e instead of (QUOTE e).

E.g.: We'd write **'(+ 3 4)** instead of **(QUOTE (+ 3 4))**.

If F is the name of some Lisp function, then:

- Evaluation of **(F '(+ 3 4))** passes the list **(+ 3 4)** as argument to a call of the function F.
- Evaluation of **(F (+ 3 4))** passes the number **7** as argument to a call of F.
- Evaluation of **(F 'X)** passes the symbol **X** as argument to a call of F.
- Evaluation of **(F X)** passes the *value of the variable denoted by* **X** as argument to a call of F.

- Numbers, NIL, and T evaluate to themselves, so you don't need to QUOTE them!
- As a matter of style, you should ***not*** type any spaces between ' and the S-expression *e* when you type '*e*.
- Don't confuse the ` and ' characters:
  ` is ***this*** character! ' is ***this*** character!

- ` can do things ' can't. It's useful for writing macros, but you won't need to use ` in this course.

Notation: For S-expressions *e* and *e'*, we write
e ⇒ *e'* to mean that *e* evaluates to *e'*.
Examples: (+ 4 5) ⇒ **9** (SQRT (+ 4 5.0)) ⇒ **3.0**
'(SQRT (3 SQRT)) ⇒ **(SQRT (3 SQRT))**

- If *e* ⇒ a nonempty list L, then:
  (**CAR** *e*) ⇒ the *first* element of that list L.

  Examples: (CAR '(DOG CAT (AT (3 +)))) ⇒ DOG
  (CAR '(DOG)) ⇒ DOG
  (CAR '((AT (3 +)) DOG CAT)) ⇒ (AT (3 +))

- If *e* ⇒ a nonempty list L, then:
  (**CDR** *e*) ⇒ the list obtained from L
  by *omitting* its first element.

  Examples: (CDR '(DOG CAT (AT (3 +)))) ⇒ (CAT (AT (3 +)))
  (CDR '(DOG)) ⇒ NIL
  (CDR '((AT (3 +)) DOG CAT)) ⇒ (DOG CAT)
  (CDR (CAR '((A B C) D E))) ⇒ (B C)
  (CAR (CDR '((A B C) D E))) ⇒ D

- If *e* ⇒ a nonempty list L, then:
  (CAR *e*) ⇒ the ***first*** element of that list L.
  Example: (CAR '(DOG CAT (AT (3 +)))) ⇒ DOG
- If *e* ⇒ a nonempty list L, then:
  (CDR *e*) ⇒ the list obtained from L
  by ***omitting*** its first element.
  Example: (CDR '(DOG CAT (AT (3 +)))) ⇒ (CAT (AT (3 +)))
- (CAR NIL) ⇒ NIL and (CDR NIL) ⇒ NIL.
  This is illogical, but sometimes convenient!
  - In *Scheme*, the car and cdr of an empty list are ***undefined***.
- If *e* ⇒ an atom other than NIL, then evaluation of (CAR *e*) or (CDR *e*) will produce an ***error***!

  E.g.: We get an ***error*** if (CAR (+ 3 4)), (CDR (+ 3 4)), (CAR (CAR '(A B))), or (CDR (CAR '(A B))) is evaluated!

Q. What happens if (CAR '(+ 3 4)) is evaluated?
What happens if (CDR '(+ 3 4)) is evaluated?

A. (CAR '(+ 3 4)) ⇒ + (CDR '(+ 3 4)) ⇒ (3 4)

- If *e* ⇒ a nonempty list L, then:
  (CAR *e*) ⇒ the *first* element of that list L.
- If *e* ⇒ a nonempty list L, then:
  (CDR *e*) ⇒ the list obtained from L
  by *omitting* its first element.
- (CAR NIL) ⇒ NIL and (CDR NIL) ⇒ NIL.
- If *e* ⇒ an atom other than NIL, then evaluation of (CAR *e*) or (CDR *e*) will produce an ***error***!

Alternative Names for CAR and CDR

- **FIRST** is another name for **CAR** in Common Lisp.
- **REST** is another name for **CDR** in Common Lisp.

Thus: (FIRST *e*) = (CAR *e*) (REST *e*) = (CDR *e*)

The names FIRST and REST have the advantage of being descriptive, but "CAR" and "CDR" provide the basis for the C…R function names we will consider later.

The following remarks regarding the origin of the names **CAR** and **CDR** are from pp. 42 – 3 of Touretzky:

These names are relics from the early days of computing, when Lisp first ran on a machine called the IBM 704. The 704 was so primitive it didn't even have transistors—it used vacuum tubes. Each of its ''registers'' was divided into several components, two of which were the address portion and the decrement portion. Back then, the name CAR stood for Contents of Address portion of Register, and CDR stood for Contents of Decrement portion of Register. Even though these terms don't apply to modern computer hardware, Common Lisp still uses the acronyms CAR and CDR when referring to cons cells, partly for historical reasons, and partly because these names can be composed to form longer names such as CADR and CDDAR, as you will see shortly.