Common Lisp S-Expressions: Atoms and Lists

Common Lisp S-expressions

The textual expressions used in Lisp code are called ***S-expressions***; S stands for "symbolic".

Common Lisp S-expressions

The textual expressions used in Lisp code are called ***S-expressions***; S stands for "symbolic".

- There are two kinds of S-expression: and

Common Lisp S-expressions

The textual expressions used in Lisp code are called ***S-expressions***; S stands for "symbolic".

- There are two kinds of S-expression: ***atoms*** and

Common Lisp S-expressions

The textual expressions used in Lisp code are called ***S-expressions***; S stands for "symbolic".

- There are two kinds of S-expression: ***atoms*** and ***lists***

Common Lisp S-expressions

The textual expressions used in Lisp code are called ***S-expressions***; S stands for "symbolic".

- There are two kinds of S-expression: ***atoms*** and ***lists***
  The *empty list* is ***both*** a list *and* an atom; it is the ***only*** S-expression that's both a list and an atom.

Common Lisp S-expressions

The textual expressions used in Lisp code are called ***S-expressions***; S stands for "symbolic".

- There are two kinds of S-expression: ***atoms*** and ***lists*** The *empty list* is ***both*** a list *and* an atom; it is the ***only*** S-expression that's both a list and an atom. The empty list can be written as   or as   .

Common Lisp S-expressions

The textual expressions used in Lisp code are called **S-*expressions***; S stands for "symbolic".

- There are two kinds of S-expression: ***<u>atoms</u>*** and ***<u>lists</u>***
  The *empty list* is ***<u>both</u>*** a list *and* an atom; it is the ***<u>only</u>*** S-expression that's both a list and an atom.
  The empty list can be written as **()** or as .

Common Lisp S-expressions

The textual expressions used in Lisp code are called ***S-expressions***; S stands for "symbolic".

- There are two kinds of S-expression: ***atoms*** and ***lists***
  The *empty list* is ***both*** a list *and* an atom; it is the ***only*** S-expression that's both a list and an atom.
  The empty list can be written as **()** or as **NIL**.

Common Lisp S-expressions

The textual expressions used in Lisp code are called ***S-expressions***; S stands for "symbolic".

- There are two kinds of S-expression: ***<u>atoms</u>*** and ***<u>lists</u>***
  The *empty list* is ***<u>both</u>*** a list *and* an atom; it is the ***<u>only</u>*** S-expression that's both a list and an atom.
  The empty list can be written as **()** or as **NIL**.
- The kinds of atom we will use in this course are:
    -
    -

Common Lisp S-expressions

The textual expressions used in Lisp code are called ***S-expressions***; S stands for "symbolic".

- There are two kinds of S-expression: ***atoms*** and ***lists***

  The *empty list* is ***both*** a list *and* an atom; it is the ***only*** S-expression that's both a list and an atom.

  The empty list can be written as **()** or as **NIL**.
- The kinds of atom we will use in this course are:
    - **Numbers** [e.g., 129, -45.33, 72.1e-4, 67/4]
    -

Common Lisp S-expressions

The textual expressions used in Lisp code are called ***S-expressions***; S stands for "symbolic".

- There are two kinds of S-expression: ***atoms*** and ***lists***
  The *empty list* is ***both*** a list *and* an atom; it is the ***only*** S-expression that's both a list and an atom.
  The empty list can be written as **()** or as **NIL**.
- The kinds of atom we will use in this course are:
    - **Numbers** [e.g., 129, -45.33, 72.1e-4, 67/4]
    - **Symbols** [e.g., X, DOG, APPLE23, NIL, FACTORIAL]

Common Lisp S-expressions

The textual expressions used in Lisp code are called ***S-expressions***; S stands for "symbolic".

- There are two kinds of S-expression: ***atoms*** and ***lists***
  The *empty list* is ***both*** a list *and* an atom; it is the ***only*** S-expression that's both a list and an atom.
  The empty list can be written as **()** or as **NIL**.
- The kinds of atom we will use in this course are:
  - **Numbers** [e.g., 129, -45.33, 72.1e-4, 67/4]
  - **Symbols** [e.g., X, DOG, APPLE23, NIL, FACTORIAL]
  - Strings [e.g., "asn.txt"]
    The only strings we will use will be filenames.

Common Lisp S-expressions

The textual expressions used in Lisp code are called ***S-expressions***; S stands for "symbolic".

- There are two kinds of S-expression: ***atoms*** and ***lists***
  The *empty list* is ***both*** a list *and* an atom; it is the ***only*** S-expression that's both a list and an atom.
  The empty list can be written as **()** or as **NIL**.
- The kinds of atom we will use in this course are:
    - **Numbers** [e.g., 129, -45.33, 72.1e-4, 67/4]
    - **Symbols** [e.g., X, DOG, APPLE23, NIL, FACTORIAL]
    - Strings [e.g., "asn.txt"]
      The only strings we will use will be filenames.
- There are two kinds of list:
    -
    -

Common Lisp S-expressions

The textual expressions used in Lisp code are called ***S-expressions***; S stands for "symbolic".

- There are two kinds of S-expression: ***atoms*** and ***lists***
  The *empty list* is ***both*** a list *and* an atom; it is the ***only*** S-expression that's both a list and an atom.
  The empty list can be written as **()** or as **NIL**.
- The kinds of atom we will use in this course are:
  - **Numbers** [e.g., 129, -45.33, 72.1e-4, 67/4]
  - **Symbols** [e.g., X, DOG, APPLE23, NIL, FACTORIAL]
  - Strings [e.g., "asn.txt"]
    The only strings we will use will be filenames.
- There are two kinds of list:
  - **Proper Lists** [e.g., (GO (X (AT) 17) (HA Y) B)]
  -

Common Lisp S-expressions

The textual expressions used in Lisp code are called ***S-expressions***; S stands for "symbolic".

- There are two kinds of S-expression: ***atoms*** and ***lists***
  The *empty list* is ***both*** a list *and* an atom; it is the ***only*** S-expression that's both a list and an atom.
  The empty list can be written as **()** or as **NIL**.
- The kinds of atom we will use in this course are:
  - **Numbers** [e.g., 129, -45.33, 72.1e-4, 67/4]
  - **Symbols** [e.g., X, DOG, APPLE23, NIL, FACTORIAL]
  - Strings [e.g., "asn.txt"]
    The only strings we will use will be filenames.
- There are two kinds of list:
  - **Proper Lists** [e.g., (GO (X (AT) 17) (HA Y) B)]
  - Dotted Lists [e.g., (GO (X (AT) 17) (HA Y) . B)]

Common Lisp S-expressions

The textual expressions used in Lisp code are called ***S-expressions***; S stands for "symbolic".

- There are two kinds of S-expression: ***atoms*** and ***lists***
  The *empty list* is ***both*** a list *and* an atom; it is the ***only*** S-expression that's both a list and an atom.
  The empty list can be written as **()** or as **NIL**.
- The kinds of atom we will use in this course are:
  - **Numbers** [e.g., 129, -45.33, 72.1e-4, 67/4]
  - **Symbols** [e.g., X, DOG, APPLE23, NIL, FACTORIAL]
  - Strings [e.g., "asn.txt"]
    The only strings we will use will be filenames.
- There are two kinds of list:
  - **Proper Lists** [e.g., (GO (X (AT) 17) (HA Y) B)]
  - Dotted Lists [e.g., (GO (X (AT) 17) (HA Y) . B)]

  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.

- Each S-expression is a textual representation of a Lisp data object that's ***also*** called an S-expression.

- Each S-expression is a textual representation of a Lisp data object that's *also* called an S-expression.
- Similarly, the Lisp data objects that are represented by atoms, numbers, symbols, strings, and proper/dotted lists are *also* called atoms, numbers, symbols, strings and proper/dotted lists.

- Each S-expression is a textual representation of a Lisp data object that's ***also*** called an S-expression.
- Similarly, the Lisp data objects that are represented by atoms, numbers, symbols, strings, and proper/dotted lists are ***also*** called atoms, numbers, symbols, strings and proper/dotted lists.

For example, the (proper) list

```
((BLUE SKY) (GREEN GRASS) (BROWN EARTH))
```

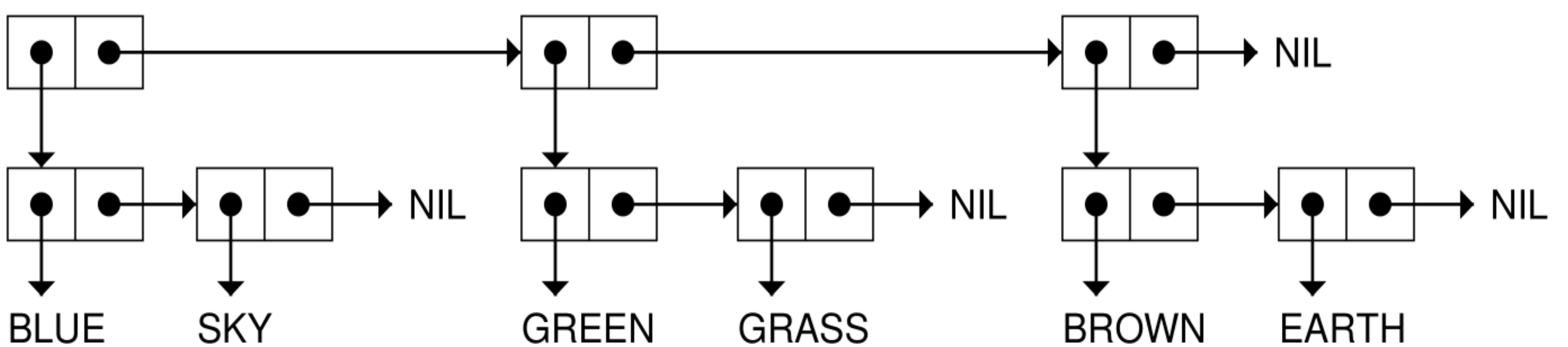
is a textual representation of a Lisp data object, ***also*** called a (proper) list, that is depicted as follows on p. 34 of Touretzky:

- Each S-expression is a textual representation of a Lisp data object that's *also* called an S-expression.
- Similarly, the Lisp data objects that are represented by atoms, numbers, symbols, strings, and proper/dotted lists are *also* called atoms, numbers, symbols, strings and proper/dotted lists.

For example, the (proper) list

    ((BLUE SKY) (GREEN GRASS) (BROWN EARTH))

is a textual representation of a Lisp data object, *also* called a (proper) list, that is depicted as follows on p. 34 of Touretzky:



S-expressions are used:

1.

2.

S-expressions are used:

1. As Lisp **code**--i.e., expressions that can be evaluated.

2.

S-expressions are used:

1. As Lisp **code**--i.e., expressions that can be evaluated. For example,

```
(sqrt (+ (* 3 2) (- 4 1)))
```

   is an S-expression that can be evaluated.

   •

   •

2.

S-expressions are used:

1. As Lisp **code**--i.e., expressions that can be evaluated. For example,

   ```
   (sqrt (+ (* 3 2) (- 4 1)))
   ```

   is an S-expression that can be evaluated.

   - All Lisp code is in the form of S-expressions.
   -

2.

S-expressions are used:

1. As Lisp **code**--i.e., expressions that can be evaluated. For example,

   `(sqrt (+ (* 3 2) (- 4 1)))`

   is an S-expression that can be evaluated.

   - All Lisp code is in the form of S-expressions.
   - But most S-expressions *cannot* be regarded as Lisp code. For example, the S-expressions `((+ 2) y 5)` and `(3 x z)` *cannot* be evaluated.

2.

S-expressions are used:

1. As Lisp **code**--i.e., expressions that can be evaluated. For example,

   `(sqrt (+ (* 3 2) (- 4 1)))`

   is an S-expression that can be evaluated.

   - All Lisp code is in the form of S-expressions.
   - But most S-expressions *cannot* be regarded as Lisp code. For example, the S-expressions `((+ 2) y 5)` and `(3 x z)` *cannot* be evaluated.

2. As Lisp **data**. Example: `((john smith) (2001 06 13))`
   In this course:

   -
   -

S-expressions are used:

1. As Lisp **code**--i.e., expressions that can be evaluated. For example,

   `(sqrt (+ (* 3 2) (- 4 1)))`

   is an S-expression that can be evaluated.

   - All Lisp code is in the form of S-expressions.
   - But most S-expressions *cannot* be regarded as Lisp code. For example, the S-expressions `((+ 2) y 5)` and `(3 x z)` *cannot* be evaluated.

2. As Lisp **data.** Example: `((john smith) (2001 06 13))`
   In this course:

   - If a Lisp variable has a value, then the value will usually be an S-expression.
   -

S-expressions are used:

1. As Lisp **code**--i.e., expressions that can be evaluated. For example,

   `(sqrt (+ (* 3 2) (- 4 1)))`

   is an S-expression that can be evaluated.

   - All Lisp code is in the form of S-expressions.
   - But most S-expressions *cannot* be regarded as Lisp code. For example, the S-expressions `((+ 2) y 5)` and `(3 x z)` *cannot* be evaluated.

2. As Lisp **data.** Example: `((john smith) (2001 06 13))`

   In this course:

   - If a Lisp variable has a value, then the value will usually be an S-expression.
   - More generally, if a Lisp expression can be evaluated, then its value (i.e., the result of its evaluation) will usually be an S-expression.

S-expressions are used:

1. As Lisp **code**--i.e., expressions that can be evaluated. For example,

```
(sqrt (+ (* 3 2) (- 4 1)))
```

   is an S-expression that can be evaluated.

2. As Lisp **data**. Example: `((john smith) (2001 06 13))`

S-expressions are used:

1. As Lisp **code**--i.e., expressions that can be evaluated. For example,

   ```
   (sqrt (+ (* 3 2) (- 4 1)))
   ```

   is an S-expression that can be evaluated.
2. As Lisp **data.** Example: `((john smith) (2001 06 13))`

S-expressions are used:

1. As Lisp **code**--i.e., expressions that can be evaluated. For example,

   ```
   (sqrt (+ (* 3 2) (- 4 1)))
   ```

   is an S-expression that can be evaluated.
2. As Lisp **data.** Example: `((john smith) (2001 06 13))`

- It is an important property of Lisp that ***Lisp code is in S-expression*** **form** ***and can therefore be Lisp data***.
-
-

S-expressions are used:

1. As Lisp **code**--i.e., expressions that can be evaluated. For example,

   `(sqrt (+ (* 3 2) (- 4 1)))`

   is an S-expression that can be evaluated.
2. As Lisp **data.** Example: `((john smith) (2001 06 13))`

- It is an important property of Lisp that ***Lisp code is in S-expression* form *<u>and can therefore be Lisp data</u>*.**
- This property makes it much easier for Lisp code to process other Lisp code or code of any language that is in S-expression form.
-

S-expressions are used:

1. As Lisp **code**--i.e., expressions that can be evaluated. For example,

   ```
   (sqrt (+ (* 3 2) (- 4 1)))
   ```

   is an S-expression that can be evaluated.

2. As Lisp **data.** Example: `((john smith) (2001 06 13))`

- It is an important property of Lisp that ***Lisp code is in S-expression*** **form** ***<u>and can therefore be Lisp data</u>*.**
- This property makes it much easier for Lisp code to process other Lisp code or code of any language that is in S-expression form.
- Lisp *macros*, which are used to extend the syntax of Lisp (i.e., "define new keywords"), depend on this property.
  -

S-expressions are used:

1. As Lisp **code**--i.e., expressions that can be evaluated. For example,

   `(sqrt (+ (* 3 2) (- 4 1)))`

   is an S-expression that can be evaluated.
2. As Lisp **data.** Example: `((john smith) (2001 06 13))`

- It is an important property of Lisp that ***Lisp code is in S-expression* form *and can therefore be Lisp data*.**
- This property makes it much easier for Lisp code to process other Lisp code or code of any language that is in S-expression form.
- Lisp *macros*, which are used to extend the syntax of Lisp (i.e., "define new keywords"), depend on this property.
  - Ch. 14 of Touretzky explains how to write macros, but you will ***not*** be expected to do that: All the macros you need to use will be predefined (i.e., built-in) macros.

- Many very commonly used Lisp forms (including SETF, DEFUN, COND, AND, OR, and LAMBDA forms) are macro forms that are predefined in terms of a set of 25 special operator forms that are often less convenient to use:

- Many very commonly used Lisp forms (including SETF, DEFUN, COND, AND, OR, and LAMBDA forms) are macro forms that are predefined in terms of a set of 25 special operator forms that are often less convenient to use:

From the Common Lisp Hyperspec (http://www.lispworks.com/documentation/common-lisp.html):

The set of *special operator names* is fixed in Common Lisp; no way is provided for the user to define a *special operator*. The next figure lists all of the Common Lisp *symbols* that have definitions as *special operators*.

block	let*	return-from
catch	load-time-value	setq
eval-when	locally	symbol-macrolet
flet	macrolet	tagbody
function	multiple-value-call	the
go	multiple-value-prog1	throw
if	progn	unwind-protect
labels	progv	
let	quote	

Figure 3-2. Common Lisp Special Operators

Just a few of these 25 special operators (probably just QUOTE, IF, LET, LET*, and LABELS) will be covered later. You will ***not*** be expected to know the rest!

More on Numbers, Symbols, and Lists in Common Lisp

Numbers (Numeric Atoms)

There are 4 kinds of number in Common Lisp:

Numbers (Numeric Atoms)

There are 4 kinds of number in Common Lisp:

- Integers (e.g., 3 or -8468284739390847398474784894)
  - Integers can have arbitrarily many digits!

Numbers (Numeric Atoms)

There are 4 kinds of number in Common Lisp:

- Integers (e.g., 3 or -8468284739390847398474784894)
  - Integers can have arbitrarily many digits!
- Ratios (e.g., -41/31) were described earlier. Further remarks:
  -
  -
  -

Numbers (Numeric Atoms)

There are 4 kinds of number in Common Lisp:

- Integers (e.g., 3 or -8468284739390847398474784894)
  - Integers can have arbitrarily many digits!
- Ratios (e.g., -41/31) were described earlier. Further remarks:
  - The numerator and denominator may have arbitrarily many digits; the denominator must be unsigned.
  -
  -

Numbers (Numeric Atoms)

There are 4 kinds of number in Common Lisp:

- Integers (e.g., 3 or -8468284739390847398474784894)
  - Integers can have arbitrarily many digits!
- Ratios (e.g., -41/31) were described earlier. Further remarks:
  - The numerator and denominator may have arbitrarily many digits; the denominator must be unsigned.
  - +, -, *, and / give ***exact*** results (i.e., there's no rounding error) if their arguments are rational.
  -

Numbers (Numeric Atoms)

There are 4 kinds of number in Common Lisp:

- Integers (e.g., 3 or -8468284739390847398474784894)
  - Integers can have arbitrarily many digits!
- Ratios (e.g., -41/31) were described earlier. Further remarks:
  - The numerator and denominator may have arbitrarily many digits; the denominator must be unsigned.
  - +, -, *, and / give ***exact*** results (i.e., there's no rounding error) if their arguments are rational.
  - If the denominator divides the numerator, then the number is an integer and not a ratio!

Numbers (Numeric Atoms)

There are 4 kinds of number in Common Lisp:

- Integers (e.g., 3 or -8468284739390847398474784894)
  - Integers can have arbitrarily many digits!
- Ratios (e.g., -41/31) were described earlier. Further remarks:
  - The numerator and denominator may have arbitrarily many digits; the denominator must be unsigned.
  - +, -, *, and / give ***exact*** results (i.e., there's no rounding error) if their arguments are rational.
  - If the denominator divides the numerator, then the number is an integer and not a ratio!
- Floating point numbers (e.g., 12.876 or 2.31e-3)
  - Common Lisp has 4 types of floating point number, (short-, single-, double-, and long-float), which differ in the amount of precision and range of exponents given by their value representations.

Numbers (Numeric Atoms)

There are 4 kinds of number in Common Lisp:

- Integers (e.g., 3 or -84682847393908473984747848 94)

- Ratios (e.g., -41/31) were described earlier.

- Floating point numbers (e.g., 12.876 or 2.31e-3)
    - Common Lisp has 4 types of floating point number, (short-, single-, double-, and long-float), which differ in the amount of precision and range of exponents given by their value representations.

Numbers (Numeric Atoms)

There are 4 kinds of number in Common Lisp:

- Integers (e.g., 3 or -8468284739390847398474784894)
- Ratios (e.g., -41/31) were described earlier.
- Floating point numbers (e.g., 12.876 or 2.31e-3)
  - Common Lisp has 4 types of floating point number, (short-, single-, double-, and long-float), which differ in the amount of precision and range of exponents given by their value representations.

Numbers (Numeric Atoms)

There are 4 kinds of number in Common Lisp:

- Integers (e.g., 3 or -8468284739390847398474784894)
- Ratios (e.g., -41/31) were described earlier.
- Floating point numbers (e.g., 12.876 or 2.31e-3)
  - Common Lisp has 4 types of floating point number (short-, single-, double-, and long-float), which differ in the amount of precision and range of exponents given by their value representations.
  - ***Single-float*** is the *default* floating point type; 12.876 and 2.31e-3 are of type single-float. You will ***not*** need to use other floating point types.

Numbers (Numeric Atoms)

There are 4 kinds of number in Common Lisp:

- Integers (e.g., 3 or -8468284739390847398474784894)
- Ratios (e.g., -41/31) were described earlier.
- Floating point numbers (e.g., 12.876 or 2.31e-3)
    - Common Lisp has 4 types of floating point number (short-, single-, double-, and long-float), which differ in the amount of precision and range of exponents given by their value representations.
    - ***Single-float*** is the *default* floating point type; 12.876 and 2.31e-3 are of type single-float. You will ***not*** need to use other floating point types
    - In Clisp, numbers of type ***single-float*** are 32-bit floating point numbers that are analogous to (i.e., that have the same value representation as) numbers of type **float** in Java.

Numbers (Numeric Atoms)

There are 4 kinds of number in Common Lisp:

- Integers (e.g., 3 or -8468284739390847398474784894)
- Ratios (e.g., -41/31) were described earlier.
- Floating point numbers (e.g., 12.876 or 2.31e-3)
  - Common Lisp has 4 types of floating point number (short-, single-, double-, and long-float), which differ in the amount of precision and range of exponents given by their value representations.
  - ***Single-float*** is the *default* floating point type; 12.876 and 2.31e-3 are of type single-float. You will ***not*** need to use other floating point types.
  - In Clisp, numbers of type ***single-float*** are 32-bit floating point numbers that are analogous to (i.e., that have the same value representation as) numbers of type **float** in Java.
  - Note that the default floating point type in Java/C++ is **double** (64 bits) rather than **float** (32 bits).

Numbers (Numeric Atoms)

There are 4 kinds of number in Common Lisp:

- Integers (e.g., 3 or -8468284739390847398474784894)
- Ratios (e.g., -41/31) were described earlier.
- Floating point numbers (e.g., 12.876 or 2.31e-3)

Numbers (Numeric Atoms)

There are 4 kinds of number in Common Lisp:

- Integers (e.g., 3 or -8468284739390847398474784894)
- Ratios (e.g., -41/31) were described earlier.
- Floating point numbers (e.g., 12.876 or 2.31e-3)
- Complex numbers (e.g., #C(3 -5/8) or #C(2.3 7.2))
    -
    -

Numbers (Numeric Atoms)

There are 4 kinds of number in Common Lisp:

- Integers (e.g., 3 or -84682847393908473984747848 94)
- Ratios (e.g., -41/31) were described earlier.
- Floating point numbers (e.g., 12.876 or 2.31e-3)
- Complex numbers (e.g., #C(3 -5/8) or #C(2.3 7.2))
    - #C(x y) represents x + yi, where i = $\sqrt{-1}$.
    -

Numbers (Numeric Atoms)

There are 4 kinds of number in Common Lisp:

- Integers (e.g., 3 or -8468284739390847398474784894)
- Ratios (e.g., -41/31) were described earlier.
- Floating point numbers (e.g., 12.876 or 2.31e-3)
- Complex numbers (e.g., #C(3 -5/8) or #C(2.3 7.2))
  - #C(x y) represents x + yi, where i = $\sqrt{-1}$.
  - We will ***not*** use complex numbers in this course!

Numbers (Numeric Atoms)

There are 4 kinds of number in Common Lisp:

- Integers (e.g., 3 or -84682847393908473984747848494)
- Ratios (e.g., -41/31) were described earlier.
- Floating point numbers (e.g., 12.876 or 2.31e-3)
- Complex numbers (e.g., #C(3 -5/8) or #C(2.3 7.2))
  - #C(x y) represents x + yi, where i = $\sqrt{-1}$.
  - We will ***not*** use complex numbers in this course!

- Numbers are one of the two kinds of Common Lisp atom that will be used extensively in this course.

  Symbols, which we consider next, are the other kind of atom we will use extensively.

Symbols (Symbolic Atoms)

Page 7 of Touretzky defines symbols as follows:

symbol Any sequence of letters, digits, and permissible special characters that is not a number.

So FOUR is a symbol, 4 is an integer, +4 is an integer, but + is a symbol. And 7−11 is also a symbol.

Symbols (Symbolic Atoms)

Page 7 of Touretzky defines symbols as follows:

symbol Any sequence of letters, digits, and permissible special characters that is not a number.

So FOUR is a symbol, 4 is an integer, +4 is an integer, but + is a symbol. And 7−11 is also a symbol.

Q. Which special characters are "permissible"?

Symbols (Symbolic Atoms)

Page 7 of Touretzky defines symbols as follows:

> **symbol** Any sequence of letters, digits, and permissible special characters that is not a number.
>
> So FOUR is a symbol, 4 is an integer, +4 is an integer, but + is a symbol. And 7−11 is also a symbol.

Q. Which special characters are "permissible"?

A. Any character that is ***not*** a whitespace character and also is ***not*** one of **( ) ' ` " , ; | \ :** is permissible, but # can't be the *first* character of a symbol and a symbol can't consist only of . characters.

Symbols (Symbolic Atoms)

Page 7 of Touretzky defines symbols as follows:

symbol Any sequence of letters, digits, and permissible special characters that is not a number.

So FOUR is a symbol, 4 is an integer, +4 is an integer, but + is a symbol. And 7−11 is also a symbol.

Q. Which special characters are "permissible"?

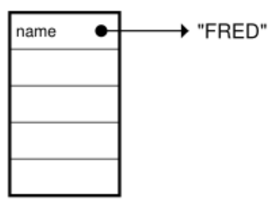
A. Any character that is ***not*** a whitespace character and also is ***not*** one of ( ) ' ` " , ; | \ : is permissible, but # can't be the *first* character of a symbol and a symbol can't consist only of . characters.

Symbols as defined here are also called ***symbol names***.

- The data object represented by a symbol name (see pp. 105-6) is called a symbol too, so use of the term *symbol name* may avoid confusion of the two concepts.

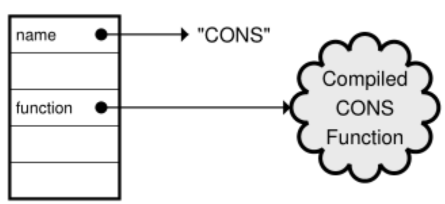
3.18 INTERNAL STRUCTURE OF SYMBOLS

So far in this book we have been drawing symbols by writing their names. But symbols in Common Lisp are actually composite objects, meaning they have several parts to them. Conceptually, a symbol is a block of five pointers, one of which points to the representation of the symbol's name. The others will be defined later. The internal structure of the symbol FRED looks like this:

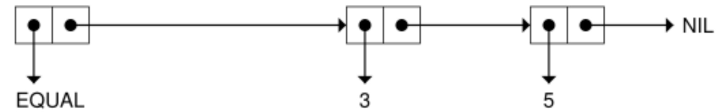


The ''FRED'' appearing above in quotation marks is called a **string**. Strings are sequences of characters; they will be covered more fully in Chapter 9. For now it suffices to note that strings are used to store the names of symbols; a symbol and its name are actually two different things.

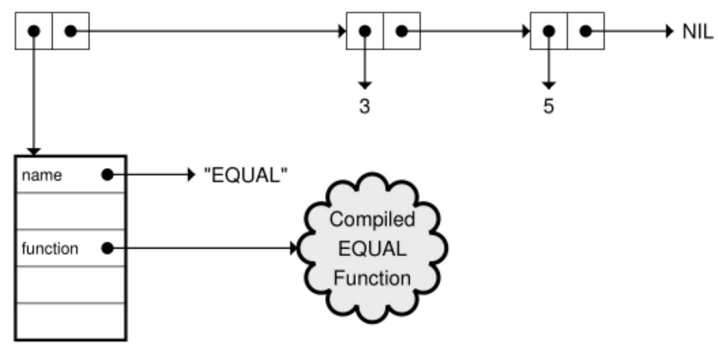
Some symbols, like CONS or +, are used to name built-in Lisp functions. The symbol CONS has a pointer in its **function cell** to a ''compiled code object'' that represents the machine language instructions for creating new cons cells.



When we draw Lisp expressions such as (EQUAL 3 5) as cons cell chains, we usually write just the name of the symbol instead of showing its internal structure:



But if we choose we can show more detail, in which case the expression (EQUAL 3 5) looks like this:



We can extract the various components of a symbol using built-in Common Lisp functions like SYMBOL-NAME and SYMBOL-FUNCTION. The following dialog illustrates this; you'll see something slightly different if you try it on your computer, but the basic idea is the same.

```
> (symbol-name 'equal)
"EQUAL"

> (symbol-function 'equal)
#<Compiled EQUAL function {60463B0}>
```

Note: Symbols are memory unique; see sec. 6.13.

Symbol names are used for the following purposes:

-
-
-

Symbol names are used for the following purposes:

- They are names of ***variables/parameters*** and *constants*.
-
-

Symbol names are used for the following purposes:

- They are names of ***variables/parameters*** and *constants*.
- They are names of ***functions*** (e.g., +, -, SQRT).
-

Symbol names are used for the following purposes:

- They are names of ***variables/parameters*** and *constants*.
- They are names of ***functions*** (e.g., +, -, SQRT).
- They are names of ***special operators*** (also called *special functions*) and ***macros***.
  -
  -
  -
  -
  -

Symbol names are used for the following purposes:

- They are names of ***variables/parameters*** and *constants*.
- They are names of ***functions*** (e.g., +, -, SQRT).
- They are names of ***special operators*** (also called *special functions*) and ***macros***.
  - 5 *special operators* that will used in this course are IF, QUOTE, LET, LET*, and LABELS.
  -
  -
  -
  -

Symbol names are used for the following purposes:

- They are names of ***variables/parameters*** and *constants*.
- They are names of ***functions*** (e.g., +, -, SQRT).
- They are names of ***special operators*** (also called *special functions*) and ***macros***.
  - 5 *special operators* that will used in this course are IF, QUOTE, LET, LET*, and LABELS.
  - Special operator expressions are evaluated in a special way--they're ***not*** evaluated like regular function calls.
  -
  -
  -

Symbol names are used for the following purposes:

- They are names of ***variables/parameters*** and *constants*.
- They are names of ***functions*** (e.g., +, -, SQRT).
- They are names of ***special operators*** (also called *special functions*) and ***macros***.
  - 5 *special operators* that will used in this course are IF, QUOTE, LET, LET*, and LABELS.
  - Special operator expressions are evaluated in a special way--they're ***not*** evaluated like regular function calls.
  - *Macros* can be thought of as "*special operators that are defined by a Lisp programmer or, in the case of a predefined macro, can be redefined by a programmer*".
  -
  -

Symbol names are used for the following purposes:

- They are names of ***variables/parameters*** and *constants*.
- They are names of ***functions*** (e.g., +, -, SQRT).
- They are names of ***special operators*** (also called *special functions*) and ***macros***.
  - 5 *special operators* that will used in this course are IF, QUOTE, LET, LET*, and LABELS.
  - Special operator expressions are evaluated in a special way--they're ***<u>not</u>*** evaluated like regular function calls.
  - *Macros* can be thought of as "*special operators that are defined by a Lisp programmer or, in the case of a predefined macro, can be redefined by a programmer*". (But it's generally a very bad idea to redefine a predefined macro, and some Lisp implementations may not even allow redefinition of certain predefined macros!)
  -
  -

Symbol names are used for the following purposes:

- They are names of ***variables/parameters*** and *constants*.
- They are names of ***functions*** (e.g., +, -, SQRT).
- They are names of ***special operators*** (also called *special functions*) and ***macros***.
  - 5 *special operators* that will used in this course are IF, QUOTE, LET, LET*, and LABELS.
  - Special operator expressions are evaluated in a special way--they're ***not*** evaluated like regular function calls.
  - *Macros* can be thought of as "*special operators that are defined by a Lisp programmer or, in the case of a predefined macro, can be redefined by a programmer*". (But it's generally a very bad idea to redefine a predefined macro, and some Lisp implementations may not even allow redefinition of certain predefined macros!)
  - 6 predefined macros that will be used in this course are SETF, DEFUN, AND, OR, COND, and LAMBDA.
  -

Symbol names are used for the following purposes:

- They are names of ***variables/parameters*** and *constants*.
- They are names of ***functions*** (e.g., +, -, SQRT).
- They are names of ***special operators*** (also called *special functions*) and ***macros***.
  - 5 *special operators* that will used in this course are IF, QUOTE, LET, LET*, and LABELS.
  - Special operator expressions are evaluated in a special way--they're ***not*** evaluated like regular function calls.
  - *Macros* can be thought of as "*special operators that are defined by a Lisp programmer or, in the case of a predefined macro, can be redefined by a programmer*". (But it's generally a very bad idea to redefine a predefined macro, and some Lisp implementations may not even allow redefinition of certain predefined macros!)
  - 6 predefined macros that will be used in this course are SETF, DEFUN, AND, OR, COND, and LAMBDA.
  - You will not be expected to define your own macros.

Symbol names are used for the following purposes:

- They are names of ***variables/parameters*** and *constants*.
- They are names of ***functions*** (e.g., +, -, SQRT).
- They are names of ***special operators*** (also called *special functions*) and ***macros***.
  - 5 *special operators* that will used in this course are IF, QUOTE, LET, LET*, and LABELS.
  - 6 predefined macros that will be used in this course are SETF, DEFUN, AND, OR, COND, and LAMBDA.
  - You will not be expected to define your own macros.

Symbol names are used for the following purposes:

- They are names of ***variables/parameters*** and *constants*.
- They are names of ***functions*** (e.g., +, -, SQRT).
- They are names of ***special operators*** (also called *special functions*) and ***macros***.
    - 5 *special operators* that will used in this course are IF, QUOTE, LET, LET*, and LABELS.
    - 6 predefined macros that will be used in this course are SETF, DEFUN, AND, OR, COND, and LAMBDA.
    - You will not be expected to define your own macros.

Symbol names are used for the following purposes:

- They are names of ***variables/parameters*** and *constants*.
- They are names of ***functions*** (e.g., +, -, SQRT).
- They are names of ***special operators*** (also called *special functions*) and ***macros***.
  - 5 *special operators* that will used in this course are IF, QUOTE, LET, LET*, and LABELS.
  - 6 predefined macros that will be used in this course are SETF, DEFUN, AND, OR, COND, and LAMBDA.
  - You will not be expected to define your own macros.

Symbols are also used as ***data***:

-
-

Symbol names are used for the following purposes:

- They are names of ***variables/parameters*** and *constants*.
- They are names of ***functions*** (e.g., +, -, SQRT).
- They are names of ***special operators*** (also called *special functions*) and ***macros***.
  - 5 *special operators* that will used in this course are IF, QUOTE, LET, LET*, and LABELS.
  - 6 predefined macros that will be used in this course are SETF, DEFUN, AND, OR, COND, and LAMBDA.
  - You will not be expected to define your own macros.

Symbols are also used as ***data***:

- Symbol names are very often used like Java/C++ **enum** constants, but they don't need to be declared.
-

Symbol names are used for the following purposes:

- They are names of ***variables/parameters*** and *constants*.
- They are names of ***functions*** (e.g., +, -, SQRT).
- They are names of ***special operators*** (also called *special functions*) and ***macros***.
  - 5 *special operators* that will used in this course are IF, QUOTE, LET, LET*, and LABELS.
  - 6 predefined macros that will be used in this course are SETF, DEFUN, AND, OR, COND, and LAMBDA.
  - You will not be expected to define your own macros.

Symbols are also used as ***data***:

- Symbol names are very often used like Java/C++ **enum** constants, but they don't need to be declared.
- The value of a variable/parameter or of any other Lisp expression may well be a symbol.

Q. Are Common Lisp symbol names case-sensitive?

Q. Are Common Lisp symbol names case-sensitive?
A. Strictly speaking, *yes*. But ...

•

Q. Are Common Lisp symbol names case-sensitive?
A. Strictly speaking, *yes*. But ...

- When Common Lisp reads a symbol name, any lowercase letters in the name are converted to uppercase.

  Example: Dog *and* dog *are both read as* DOG *by Lisp*.

  -
  -
  -

Q. Are Common Lisp symbol names case-sensitive?
A. Strictly speaking, *yes*. But ...

- When Common Lisp reads a symbol name, any lowercase letters in the name are converted to uppercase.

  Example: Dog *and* dog *are both read as* DOG *by Lisp*.

  - This case conversion may be prevented by typing \ before each lowercase letter (as in D\o\g), or by typing the symbol name between two | characters (as in |Dog|).
  -
  -

Q. Are Common Lisp symbol names case-sensitive?
A. Strictly speaking, *yes*. But ...

- When Common Lisp reads a symbol name, any lowercase letters in the name are converted to uppercase.

  Example: Dog *and* dog *are both read as* DOG *by Lisp*.

  - This case conversion may be prevented by typing \ before each lowercase letter (as in D\o\g), or by typing the symbol name between two | characters (as in |Dog|).
  - The characters \ and | are called *escape characters*.
  -

Q. Are Common Lisp symbol names case-sensitive?
A. Strictly speaking, *yes*. But ...

- When Common Lisp reads a symbol name, any lowercase letters in the name are converted to uppercase.

  Example: Dog *and* dog *are both read as* DOG *by Lisp*.

  - This case conversion may be prevented by typing \ before each lowercase letter (as in D\o\g), or by typing the symbol name between two | characters (as in |Dog|).
  - The characters \ and | are called *escape characters*.
  - There is no such case conversion in some versions of Scheme, nor in the Racket and Clojure dialects of Lisp; symbol names in those Lisp dialects are unambiguously case-sensitive.

Q. Are Common Lisp symbol names case-sensitive?
A. Strictly speaking, *yes*. But ...

- When Common Lisp reads a symbol name, any lowercase letters in the name are converted to uppercase.

  Example: Dog *and* dog *are both read as* DOG *by Lisp*.

  - This case conversion may be prevented by typing \ before each lowercase letter (as in D\o\g), or by typing the symbol name between two | characters (as in |Dog|).
  - The characters \ and | are called *escape characters*.
  - There is no such case conversion in some versions of Scheme, nor in the Racket and Clojure dialects of Lisp; symbol names in those Lisp dialects are unambiguously case-sensitive.

  Comment: Escape characters can also be used to create symbols with names that would otherwise not be allowed. But we will not use such symbols in this course.

The Symbols NIL and T

-
-
-
-
-

The Symbols NIL and T

- **NIL** is a constant that denotes the empty list.
-
-
-
-

The Symbols NIL and T

- **NIL** is a constant that denotes the empty list.
- NIL can also be written as **()**.
-
-
-

The Symbols NIL and T

- **NIL** is a constant that denotes the empty list.
- NIL can also be written as **()**.
- NIL is also used to mean **false** in Common Lisp.
-
-

The Symbols NIL and T

- **NIL** is a constant that denotes the empty list.
- NIL can also be written as **()**.
- NIL is also used to mean **false** in Common Lisp.
- **T** is a constant that is the *usual* way to represent **true**, though any S-expression other than NIL can also be used to represent **true**.
-

The Symbols NIL and T

- **NIL** is a constant that denotes the empty list.
- NIL can also be written as **()**.
- NIL is also used to mean **false** in Common Lisp.
- **T** is a constant that is the *usual* way to represent **true**, though any S-expression other than NIL can also be used to represent **true**.
- T and NIL evaluate to themselves:
  The value of T is always T itself;
  the value of NIL is always NIL itself.

  The values of T and NIL can't be changed, and you ***can't*** use T or NIL as a variable / formal parameter!

Lists

As mentioned earlier, there are two kinds of list:
(1) proper lists (2) dotted lists

Lists

As mentioned earlier, there are two kinds of list:

(1) proper lists (2) dotted lists

- *Proper* lists of length 0, 1, 2, ..., n have the forms

  $$(),\ (e_1),\ (e_1\ e_2),\ (e_1\ e_2\ \dots\ e_n)$$

  where each e can be any S-expression (i.e., any atom or list); lists can be nested to any depth.

Lists

As mentioned earlier, there are two kinds of list:

(1) proper lists (2) dotted lists

- *Proper* lists of length 0, 1, 2, ..., n have the forms

  $$(), (e_1), (e_1\ e_2), (e_1\ e_2\ ...\ e_n)$$

  where each e can be any S-expression (i.e., any atom or list); lists can be nested to any depth.
- A *proper* list $(e_1\ \dots\ e_n)$ represents a singly-linked list data structure that may be drawn as follows:

Lists

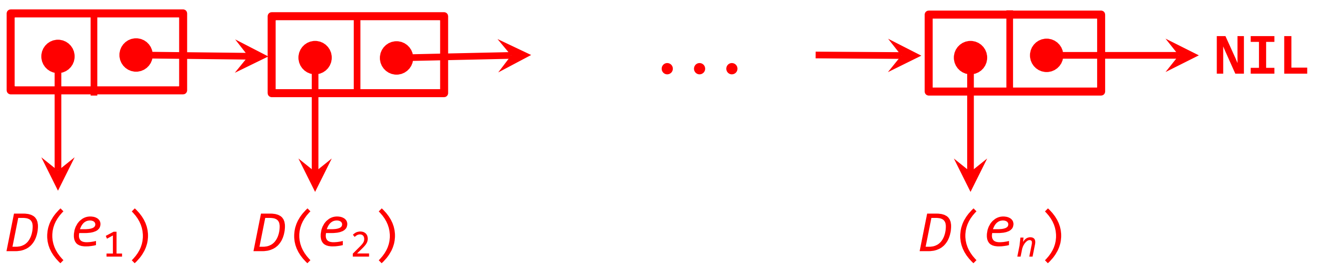
As mentioned earlier, there are two kinds of list:
(1) proper lists (2) dotted lists

- *Proper* lists of length 0, 1, 2, ..., n have the forms
  (), (e_1), (e_1 e_2), (e_1 e_2 ... e_n)
  where each e can be any S-expression (i.e., any atom or list); lists can be nested to any depth.
- A *proper* list (e_1 … e_n) represents a singly-linked list data structure that may be drawn as follows:

Here $D(e_i)$ is a drawing of the data structure represented by e_i.

Lists

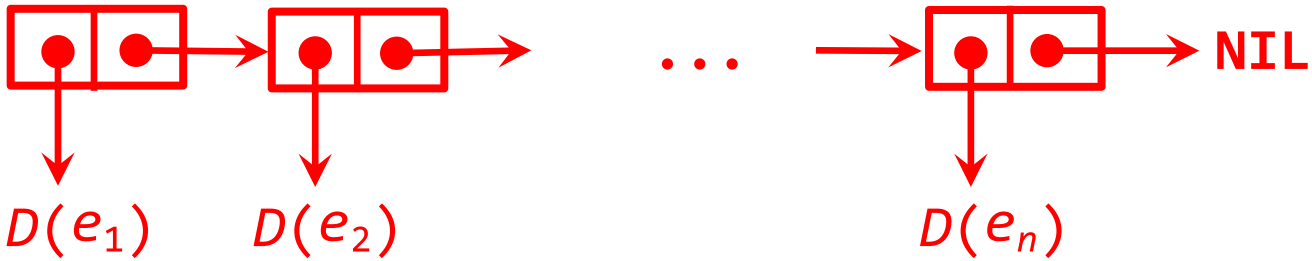
- A *proper* list (e_1 … e_n) represents a singly-linked list data structure that may be drawn as follows:



  Here $D(e_i)$ is a drawing of the data structure represented by e_i.

Lists

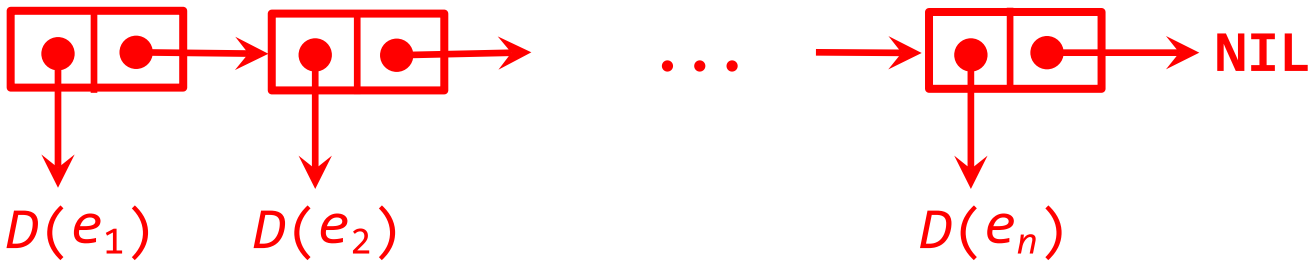
- A *proper* list (e_1 … e_n) represents a singly-linked list data structure that may be drawn as follows:



Here $D(e_i)$ is a drawing of the data structure represented by e_i.

Lists

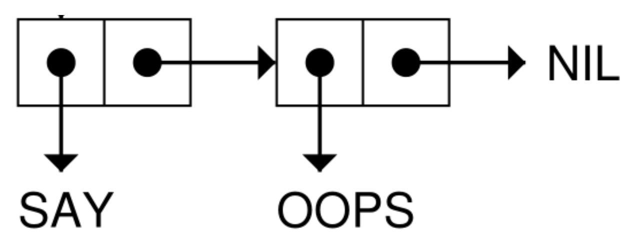
- A *proper* list (e_1 … e_n) represents a singly-linked list data structure that may be drawn as follows:



Here $D(e_i)$ is a drawing of the data structure represented by e_i.

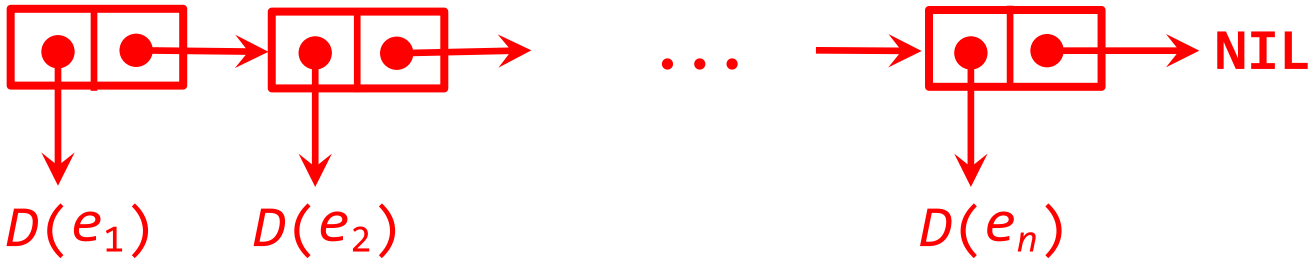
Example from p. 34 of Touretzky.

(SAY OOPS) represents:



Lists

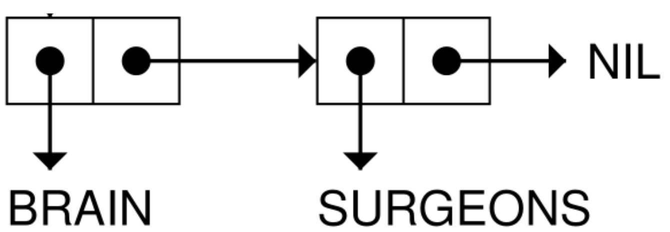
- A *proper* list (e_1 … e_n) represents a singly-linked list data structure that may be drawn as follows:



Here $D(e_i)$ is a drawing of the data structure represented by e_i.

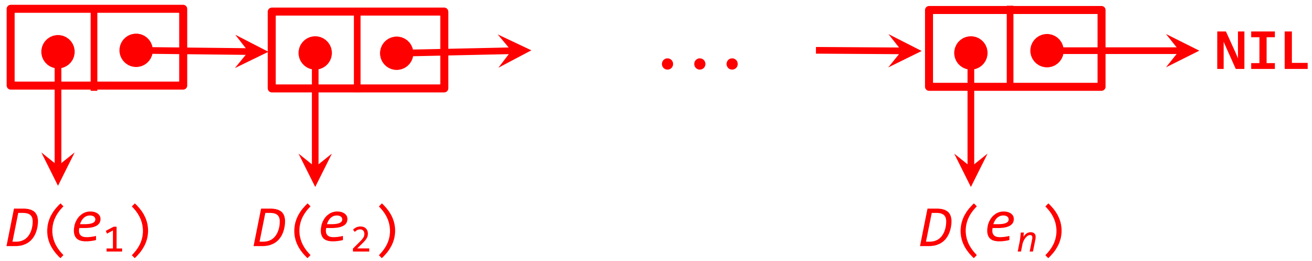
From p. 34 of Touretzky.

(BRAIN SURGEONS) represents:



Lists

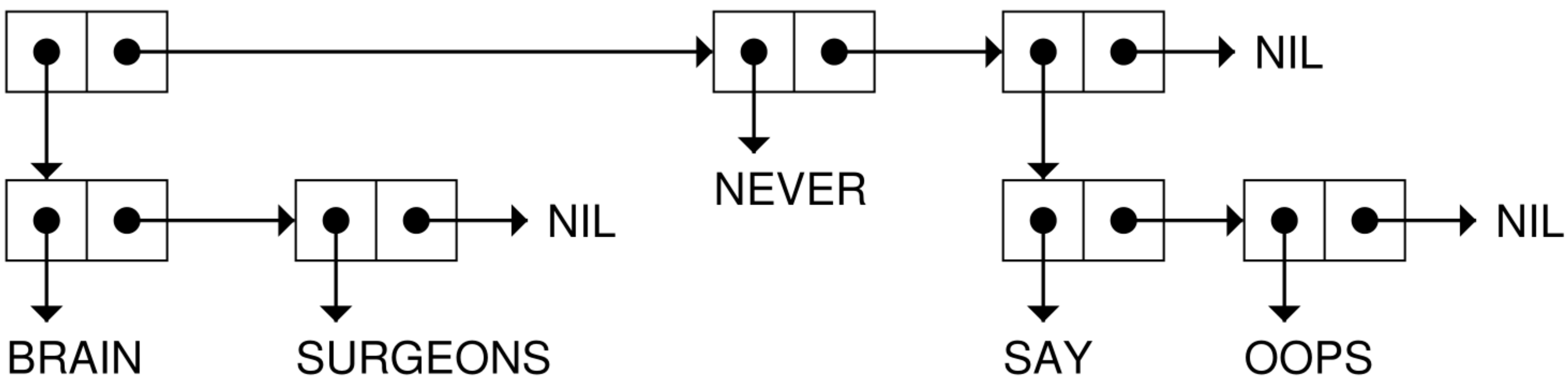
- A *proper* list (e_1 … e_n) represents a singly-linked list data structure that may be drawn as follows:



Here $D(e_i)$ is a drawing of the data structure represented by e_i.

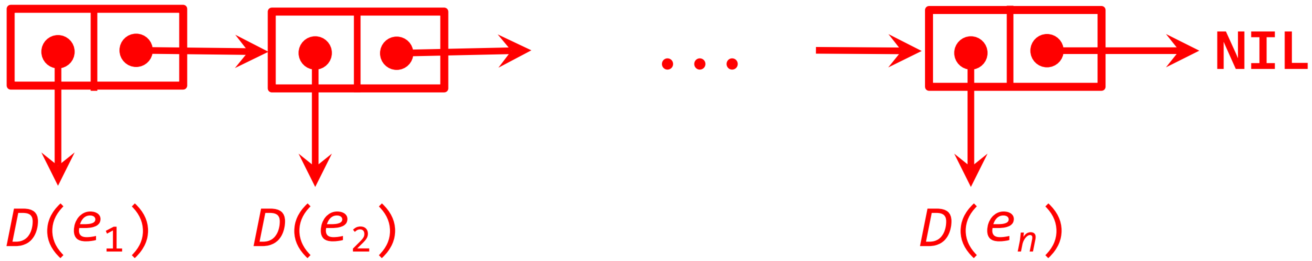
From p. 34 of Touretzky.

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



Lists

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



Here $D(e_i)$ is a drawing of the data structure represented by e_i.

Lists

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

Here $D(e_i)$ is a drawing of the data structure represented by e_i.

Lists

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

  Here $D(e_i)$ is a drawing of the data structure represented by e_i.

- A ***dotted*** list has the form (e_1 … e_n . a) where $n \geq 1$, each e is an S-expression, and a ***is an atom other than*** **NIL**.

•

•

Lists

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

  Here $D(e_i)$ is a drawing of the data structure represented by e_i.

- A ***dotted*** list has the form (e_1 … e_n . a) where $n \geq 1$, each e is an S-expression, and a <u>***is an atom other than*** **NIL**</u>.

- The notation (e_1 … e_n . a) can also be used if a is a list.

-

Lists

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

Here $D(e_i)$ is a drawing of the data structure represented by e_i.

- A ***dotted*** list has the form (e_1 … e_n . a) where $n \geq 1$, each e is an S-expression, and a ***is an atom other than*** **NIL**.

- The notation (e_1 … e_n . a) can also be used if a is a list.

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

Lists

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

  Here $D(e_i)$ is a drawing of the data structure represented by e_i.

- A ***dotted*** list has the form (e_1 … e_n . a) where $n \geq 1$, each e is an S-expression, and a ***is an atom other than* NIL**.

- The notation (e_1 … e_n . a) can also be used if a is a list.

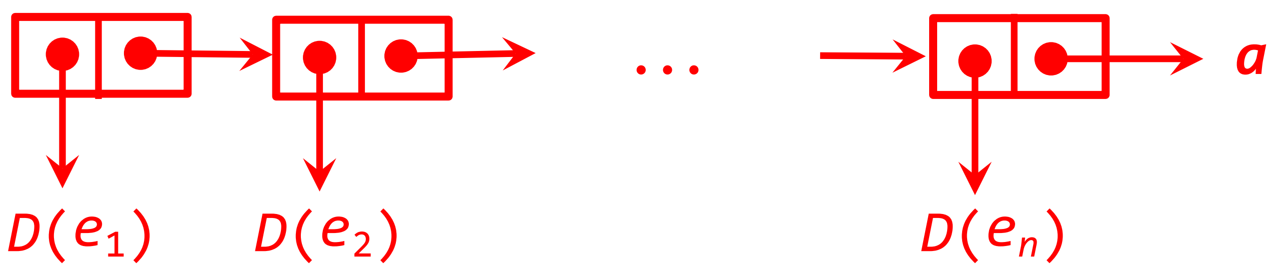
- If a is an atom, then (e_1 … e_n . a) represents a singly-linked list data structure that may be drawn as follows:

Lists

- A ***dotted*** list has the form (e_1 … e_n . a) where $n \geq 1$, each e is an S-expression, and a <u>***is an atom other than*** **NIL**</u>.
- The notation (e_1 … e_n . a) can also be used if a is a list.
- If a is an atom, then (e_1 … e_n . a) represents a singly-linked list data structure that may be drawn as follows:

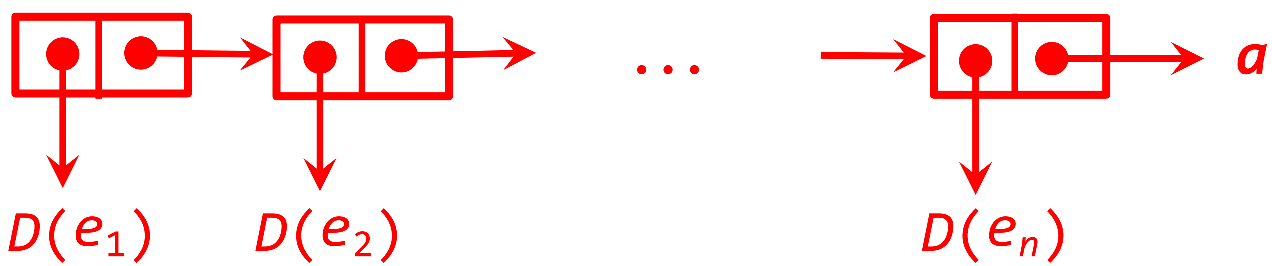
Lists

- A ***dotted*** list has the form (e_1 … e_n . a) where $n \geq 1$, each e is an S-expression, and a ***is an atom other than* NIL**.
- The notation (e_1 … e_n . a) can also be used if a is a list.
- If a is an atom, then (e_1 … e_n . a) represents a singly-linked list data structure that may be drawn as follows:



Lists

- A ***dotted*** list has the form (e_1 … e_n . a) where $n \geq 1$, each e is an S-expression, and a ***is an atom other than*** **NIL**.
- The notation (e_1 … e_n . a) can also be used if a is a list.
- If a is an atom, then (e_1 … e_n . a) represents a singly-linked list data structure that may be drawn as follows:



Examples from pp. 72-3 of Touretzky

The dotted list

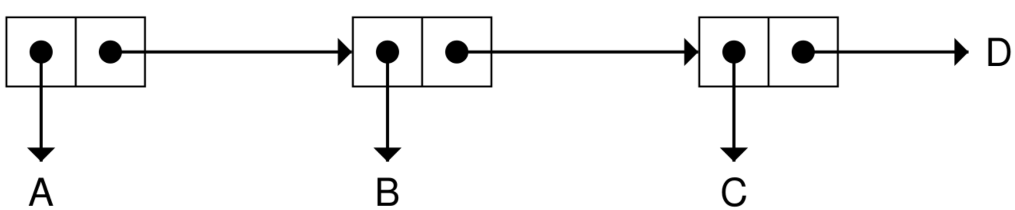
(A B C . D) represents:

Lists

- A ***dotted*** list has the form (e_1 … e_n . a) where $n \geq 1$, each e is an S-expression, and a ***is an atom other than* NIL**.
- The notation (e_1 … e_n . a) can also be used if a is a list.
- If a is an atom, then (e_1 … e_n . a) represents a singly-linked list data structure that may be drawn as follows:

Examples from pp. 72-3 of Touretzky

The dotted list
(A B C . D) represents:

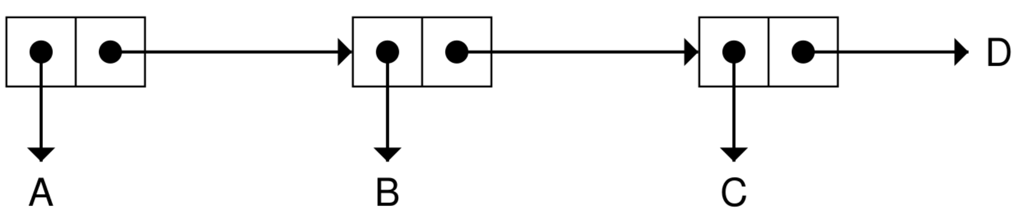


Lists

- A ***dotted*** list has the form (e_1 … e_n . a) where $n \geq 1$, each e is an S-expression, and a ***is an atom other than*** **NIL**.
- The notation (e_1 … e_n . a) can also be used if a is a list.
- If a is an atom, then (e_1 … e_n . a) represents a singly-linked list data structure that may be drawn as follows:

Examples from pp. 72-3 of Touretzky

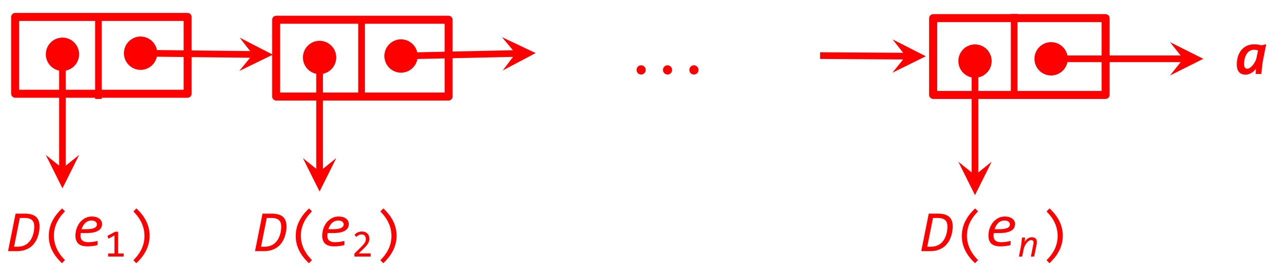
The dotted list (A B C . D) represents:



The dotted pair (A . B) represents:

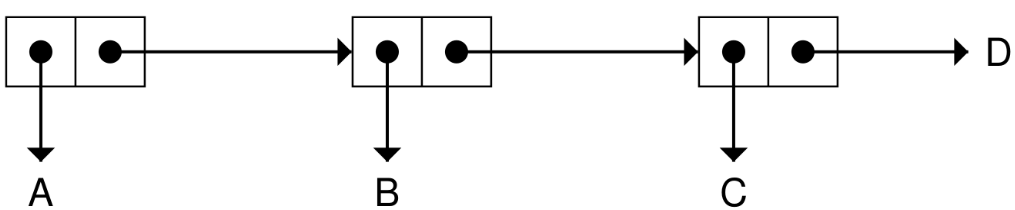
Lists

- A ***dotted*** list has the form (e_1 … e_n . a) where $n \geq 1$, each e is an S-expression, and a ***is an atom other than*** **NIL**.
- The notation (e_1 … e_n . a) can also be used if a is a list.
- If a is an atom, then (e_1 … e_n . a) represents a singly-linked list data structure that may be drawn as follows:

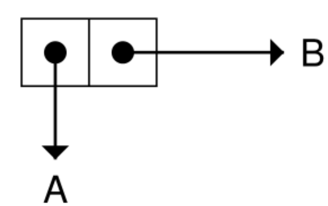


Examples from pp. 72-3 of Touretzky

The dotted list (A B C . D) represents:

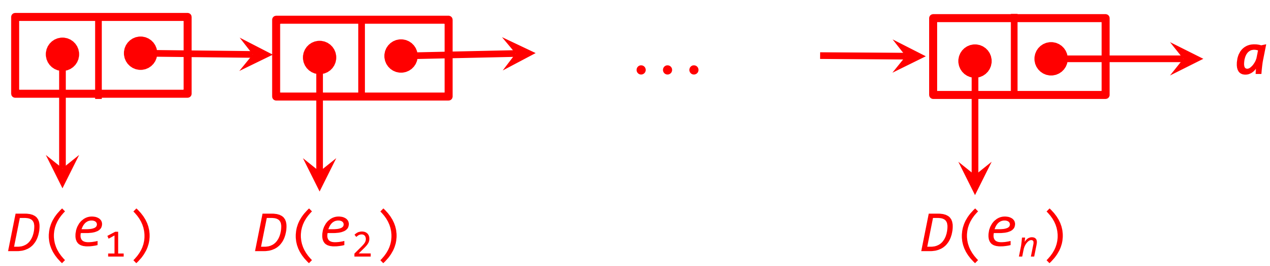


The dotted pair (A . B) represents:



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:



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:

Lists

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

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

-

-

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:

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

-

Lists

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

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

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

(e_1 … e_n . NIL) = (e_1 … e_n)
(e_1 … e_n . (e_{n+1})) =
(e_1 … e_n . (e_{n+1} … e_{n+k})) =
(e_1 … e_n . (e_{n+1} … e_{n+k} . a)) =

-

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:

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

-

Lists

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

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

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

$$(e_1 \;\dots\; e_n \;.\; \mathtt{NIL}) = (e_1 \dots e_n)$$
$$(e_1 \;\dots\; e_n \;.\; (e_{n+1})) = (e_1 \dots e_n\; e_{n+1})$$
$$(e_1 \;\dots\; e_n \;.\; (e_{n+1} \dots e_{n+k})) = (e_1 \dots e_n\; e_{n+1} \dots e_{n+k})$$
$$(e_1 \;\dots\; e_n \;.\; (e_{n+1} \dots e_{n+k} \;.\; a)) =$$

-

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:

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

-

Lists

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

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

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

(e_1 … e_n . NIL) = (e_1 … e_n)
(e_1 … e_n . (e_{n+1})) = (e_1 … e_n e_{n+1})
(e_1 … e_n . (e_{n+1} … e_{n+k})) = (e_1 … e_n e_{n+1} … e_{n+k})
(e_1 … e_n . (e_{n+1} … e_{n+k} . a)) = (e_1 … e_n e_{n+1} … e_{n+k} . a)

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

Lists

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

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

- So we'll write (e_1 … 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:

Lists

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

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

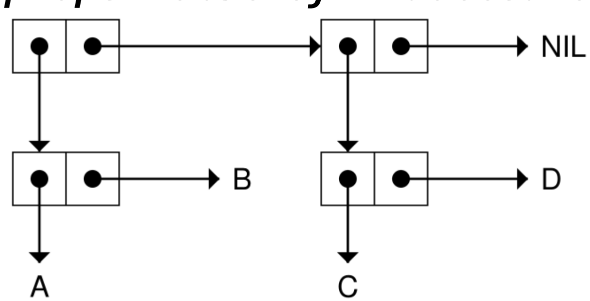
((A . B) (C . D)) is a *proper list of 2 dotted lists* that represents:

Lists

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

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

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

Further Comments on Dotted Lists

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

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

-
-
-
-

Further Comments on Dotted Lists

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

- A *proper* list (e_1 … 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 … e_n . a) is n (***not*** n+1).

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.
-
-

Further Comments on Dotted Lists

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

- A *proper* list (e_1 … 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 … 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***!
-

Further Comments on Dotted Lists

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

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

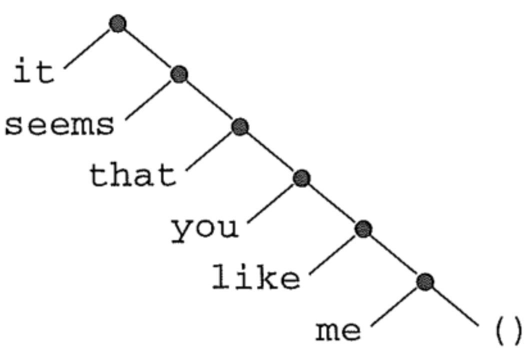
- If a is an atom, the ***length*** of (e_1 … e_n . a) is n (***not*** n+1).
- Dotted lists are used much less than proper lists.
- The term ***list*** is often used to mean ***proper list***!
- If a function you write for this course returns a dotted list, then either your code has a bug or an inappropriate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

Tree Representation of S-Expressions

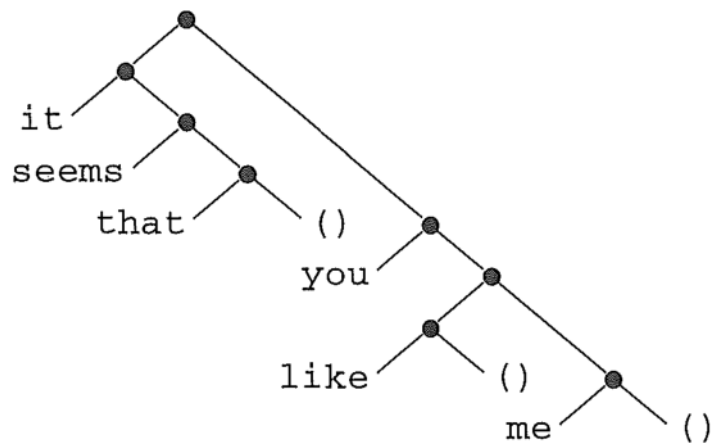
These "box and arrow" drawings are quite often simplified to ***binary trees*** in which the two children of any internal node v are ***either*** south and east of v ***or*** southwest and southeast of v:

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)

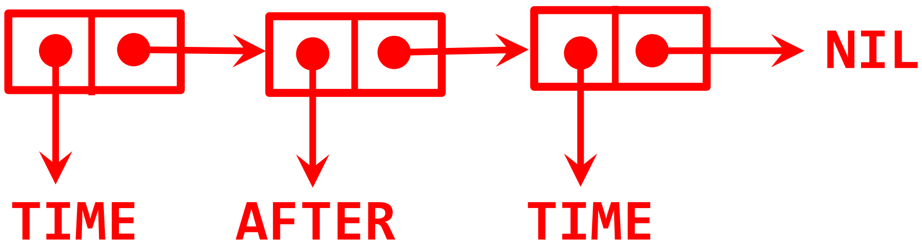
These two examples are from p. 393 of Sethi's book.

Memory Uniqueness of Symbols

(TIME AFTER TIME) represents:

Memory Uniqueness of Symbols

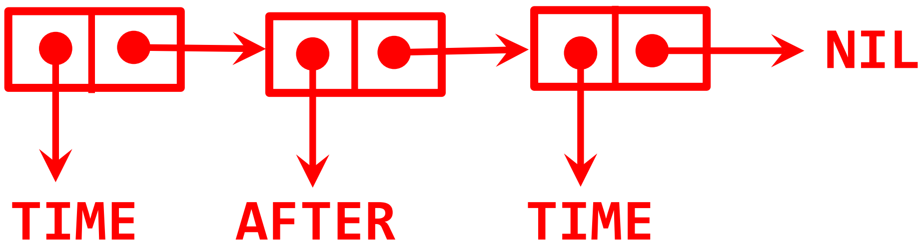
(TIME AFTER TIME) represents:



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

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.

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.

Memory Uniqueness of Symbols

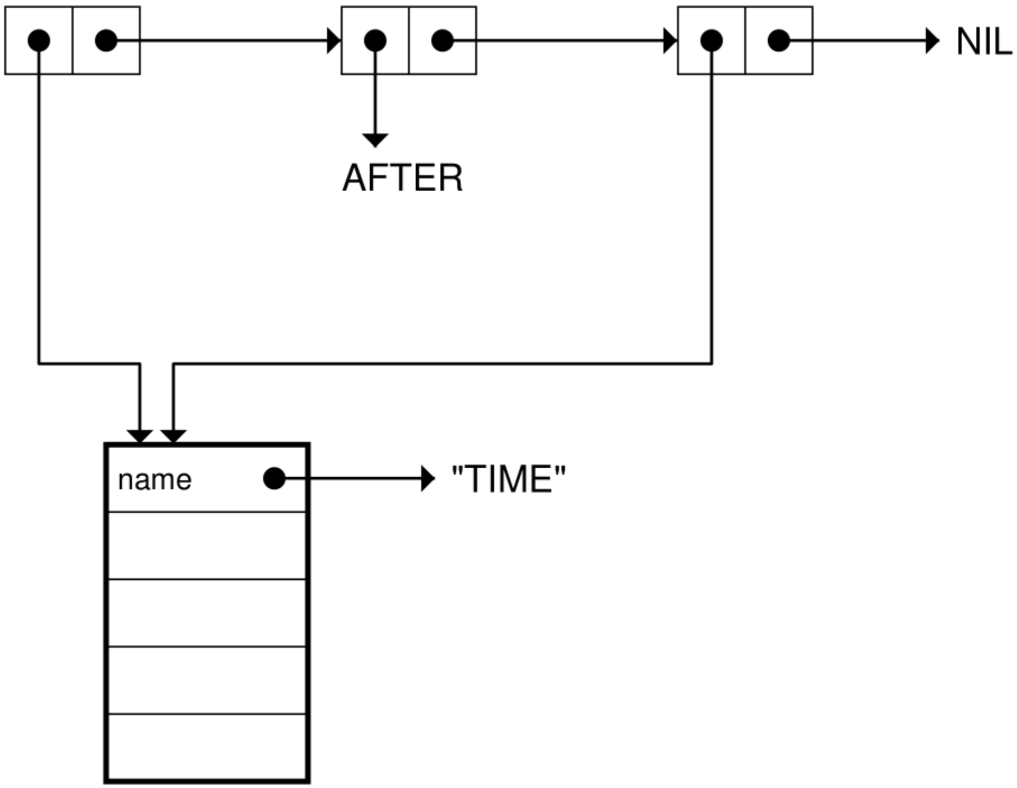
In Lisp, symbols are unique, meaning there can be only one symbol in the computer's memory with a given name.** Every object in the memory has a numbered location, called its **address**. Since a symbol exists in only one place in memory, symbols have unique addresses. So in the list (TIME AFTER TIME), 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:

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.

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***:

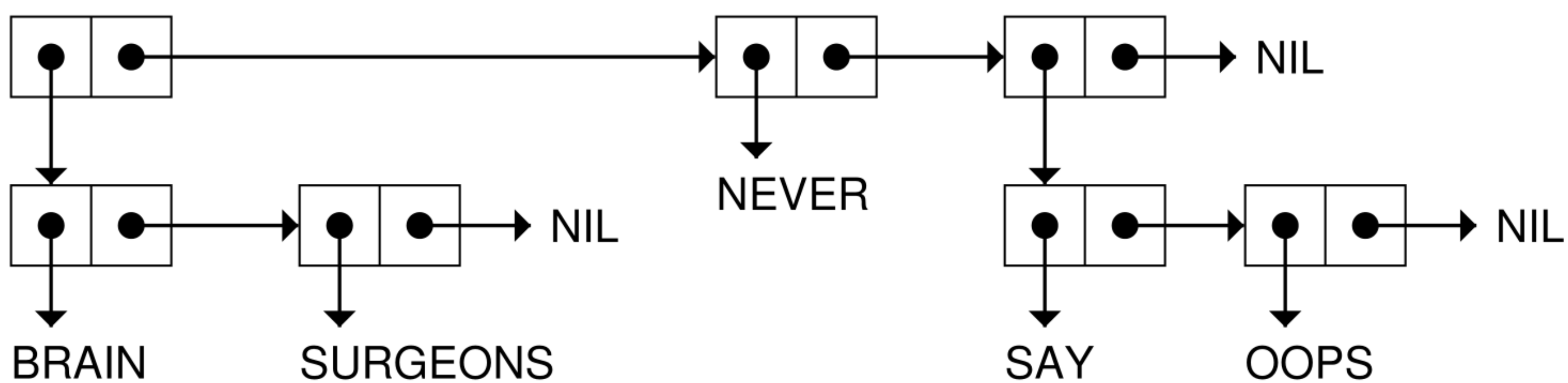
Memory Uniqueness of Symbols

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:



Evaluation of S-Expressions

- To ***evaluate*** an S-expression is to compute its ***value***; evaluation is said to ***return*** the value, and the S-expression is said to ***evaluate to*** that value.
- **Example**:
-
-
-

- To ***evaluate*** an S-expression is to compute its ***value***; evaluation is said to ***return*** the value, and the S-expression is said to ***evaluate to*** that value.
- **Example**: The S-expression **(+ 3 4)** *evaluates to* the number **7**; the number **7** is the *value* of **(+ 3 4)**.
-
-
-

- To ***evaluate*** an S-expression is to compute its ***value***; evaluation is said to ***return*** the value, and the S-expression is said to ***evaluate to*** that value.
- **Example**: The S-expression **(+ 3 4)** *evaluates to* the number **7**; the number **7** is the *value* of **(+ 3 4)**.
- When a user enters an S-expression at a Lisp prompt, the Lisp interpreter does the following:
    1.
    2.
    3.
-
-

- To ***evaluate*** an S-expression is to compute its ***value***; evaluation is said to ***return*** the value, and the S-expression is said to ***evaluate to*** that value.
- **Example**: The S-expression **(+ 3 4)** *evaluates to* the number **7**; the number **7** is the *value* of **(+ 3 4)**.
- When a user enters an S-expression at a Lisp prompt, the Lisp interpreter does the following:
  1. It ***read***s the S-expression that was entered.
  2.
  3.
-
-

- To ***evaluate*** an S-expression is to compute its ***value***; evaluation is said to ***return*** the value, and the S-expression is said to ***evaluate to*** that value.
- **Example**: The S-expression **(+ 3 4)** *evaluates to* the number **7**; the number **7** is the *value* of **(+ 3 4)**.
- When a user enters an S-expression at a Lisp prompt, the Lisp interpreter does the following:
    1. It ***read***s the S-expression that was entered.
    2. It attempts to ***evaluate*** the S-expression.
    3.
-
-

- To ***evaluate*** an S-expression is to compute its ***value***; evaluation is said to ***return*** the value, and the S-expression is said to ***evaluate to*** that value.
- **Example**: The S-expression **(+ 3 4)** *evaluates to* the number **7**; the number **7** is the *value* of **(+ 3 4)**.
- When a user enters an S-expression at a Lisp prompt, the Lisp interpreter does the following:
  1. It ***read***s the S-expression that was entered.
  2. It attempts to ***evaluate*** the S-expression.
  3. It ***print***s the value that is returned by evaluation if evaluation is successful.
-
-

- To ***evaluate*** an S-expression is to compute its ***value***; evaluation is said to ***return*** the value, and the S-expression is said to ***evaluate to*** that value.
- **Example**: The S-expression **(+ 3 4)** *evaluates to* the number **7**; the number **7** is the *value* of **(+ 3 4)**.
- When a user enters an S-expression at a Lisp prompt, the Lisp interpreter does the following:
  1. It ***read***s the S-expression that was entered.
  2. It attempts to ***evaluate*** the S-expression.
  3. It ***print***s the value that is returned by evaluation if evaluation is successful.
- Many S-expressions ***cannot*** be successfully evaluated --e.g., neither (3 4) nor (/ 2 0) can be evaluated: Any attempt to evaluate such an S-expression produces an *error*; these S-expressions have no value!
-

- To ***evaluate*** an S-expression is to compute its ***value***; evaluation is said to ***return*** the value, and the S-expression is said to ***evaluate to*** that value.
- **Example**: The S-expression **(+ 3 4)** *evaluates to* the number **7**; the number **7** is the *value* of **(+ 3 4)**.
- When a user enters an S-expression at a Lisp prompt, the Lisp interpreter does the following:
  1. It ***read***s the S-expression that was entered.
  2. It attempts to ***evaluate*** the S-expression.
  3. It ***print***s the value that is returned by evaluation if evaluation is successful.
- Many S-expressions ***cannot*** be successfully evaluated --e.g., neither (3 4) nor (/ 2 0) can be evaluated: Any attempt to evaluate such an S-expression produces an *error*; these S-expressions have no value!
- If an S-expression can be evaluated, then its value will be a data object that may be an atom or a list.

How S-Expressions are Evaluated

There are 3 cases:

How S-Expressions are Evaluated

There are 3 cases:

Evaluation of **symbols**.

How S-Expressions are Evaluated

There are 3 cases:

Evaluation of **symbols**.

Evaluation of **atoms that are not symbols**.

How S-Expressions are Evaluated

There are 3 cases:

Evaluation of **symbols**.

Evaluation of **atoms that are not symbols**.

Evaluation of **nonempty proper lists**.

How S-Expressions are Evaluated

There are 3 cases:

Evaluation of **symbols**.

Evaluation of **atoms that are not symbols**.

Evaluation of **nonempty proper lists**.

1. A **symbol** evaluates to the value of the variable / formal parameter or constant that is denoted by the symbol.

How S-Expressions are Evaluated

There are 3 cases:

Evaluation of **symbols**.

Evaluation of **atoms that are not symbols**.

Evaluation of **nonempty proper lists**.

1. A **symbol** evaluates to the value of the variable / formal parameter or constant that is denoted by the symbol.
   - If the symbol denotes a variable that has no value, then an evaluation error occurs.

How S-Expressions are Evaluated

There are 3 cases:

Evaluation of **symbols**.

Evaluation of **atoms that are not symbols**.

Evaluation of **nonempty proper lists**.

1. A **symbol** evaluates to the value of the variable / formal parameter or constant that is denoted by the symbol.
   - If the symbol denotes a variable that has no value, then an evaluation error occurs.
2. Any **atom that is not a symbol** evaluates to the same atom--e.g., ***any number evaluates to the same number***.

How S-Expressions are Evaluated

There are 3 cases:

Evaluation of **symbols**.

Evaluation of **atoms that are not symbols**.

Evaluation of **nonempty proper lists**.

1. A **symbol** evaluates to the value of the variable / formal parameter or constant that is denoted by the symbol.
   - If the symbol denotes a variable that has no value, then an evaluation error occurs.
2. Any **atom that is not a symbol** evaluates to the same atom--e.g., ***any number evaluates to the same number***.
3. A **nonempty proper list** (e_1 … e_n) may be evaluated as a ***or*** as a depending on e_1, the ***first element*** of the list.

How S-Expressions are Evaluated

There are 3 cases:

Evaluation of **symbols**.

Evaluation of **atoms that are not symbols**.

Evaluation of **nonempty proper lists**.

1. A **symbol** evaluates to the value of the variable / formal parameter or constant that is denoted by the symbol.
   - If the symbol denotes a variable that has no value, then an evaluation error occurs.
2. Any **atom that is not a symbol** evaluates to the same atom--e.g., ***any number evaluates to the same number***.
3. A **nonempty proper list** (e_1 … e_n) may be evaluated as a ***function call*** ***or*** as a depending on e_1, the ***first element*** of the list.

How S-Expressions are Evaluated

There are 3 cases:

Evaluation of **symbols**.

Evaluation of **atoms that are not symbols**.

Evaluation of **nonempty proper lists**.

1. A **symbol** evaluates to the value of the variable / formal parameter or constant that is denoted by the symbol.
   - If the symbol denotes a variable that has no value, then an evaluation error occurs.
2. Any **atom that is not a symbol** evaluates to the same atom--e.g., ***any number evaluates to the same number***.
3. A **nonempty proper list** (e_1 … e_n) may be evaluated as a ***function call*** ***or*** as a ***special/macro form*** depending on e_1, the ***first element*** of the list.

How S-Expressions are Evaluated

1. A **symbol** evaluates to the value of the variable / formal parameter or constant that is denoted by the symbol.

2. Any **atom that is not a symbol** evaluates to the same atom.

3. A **nonempty proper list** (e_1 … e_n) may be evaluated as a ***function call*** ***or*** as a ***special/macro form*** depending on e_1, the ***first element*** of the list.

How S-Expressions are Evaluated

1. A **symbol** evaluates to the value of the variable / formal parameter or constant that is denoted by the symbol.
2. Any **atom that is not a symbol** evaluates to the same atom.
3. A **nonempty proper list** (e_1 … e_n) may be evaluated as a ***function call*** ***or*** as a ***special/macro form*** depending on e_1, the ***first element*** of the list.

How S-Expressions are Evaluated

1. A **symbol** evaluates to the value of the variable / formal parameter or constant that is denoted by the symbol.
2. Any **atom that is not a symbol** evaluates to the same atom.
3. A **nonempty proper list** (e_1 … e_n) may be evaluated as a ***function call*** ***or*** as a ***special/macro form*** depending on e_1, the ***first element*** of the list.

How S-Expressions are Evaluated

1. A **symbol** evaluates to the value of the variable / formal parameter or constant that is denoted by the symbol.
2. Any **atom that is not a symbol** evaluates to the same atom.
3. A **nonempty proper list** $(e_1 \;\; \dots \;\; e_n)$ may be evaluated as a ***function call*** ***or*** as a ***special/macro form*** depending on e_1, the ***first element*** of the list:
   -
   -

How S-Expressions are Evaluated

1. A **symbol** evaluates to the value of the variable / formal parameter or constant that is denoted by the symbol.
2. Any **atom that is not a symbol** evaluates to the same atom.
3. A **nonempty proper list** (e_1 … e_n) may be evaluated as a ***function call*** ***or*** as a ***special/macro form*** depending on e_1, the ***first element*** of the list:
   - If e_1 is ***the name of a function*** or e_1 is ***a lambda expression*** (a kind of list we will study later), then:
     -
     -
   - If e_1 is ***the name of a special operator or macro***, then:
     -
     -

How S-Expressions are Evaluated

1. A **symbol** evaluates to the value of the variable / formal parameter or constant that is denoted by the symbol.
2. Any **atom that is not a symbol** evaluates to the same atom.
3. A **nonempty proper list** (e_1 … e_n) may be evaluated as a ***function call*** ***or*** as a ***special/macro form*** depending on e_1, the ***first element*** of the list:
   - If e_1 is ***the name of a function*** or e_1 is ***a lambda expression*** (a kind of list we will study later), then:
     - (e_1 … e_n) is evaluated as a ***call of the function given by*** e_1; the values obtained by evaluating e_2, ..., e_n are passed to the call as argument values.
     - **Example**:
   - If e_1 is ***the name of a special operator or macro***, then:
     -
     -

How S-Expressions are Evaluated

1. A **symbol** evaluates to the value of the variable / formal parameter or constant that is denoted by the symbol.
2. Any **atom that is not a symbol** evaluates to the same atom.
3. A **nonempty proper list** (e_1 … e_n) may be evaluated as a ***function call*** ***or*** as a ***special/macro form*** depending on e_1, the ***first element*** of the list:
   - If e_1 is ***the name of a function*** or e_1 is ***a lambda expression*** (a kind of list we will study later), then:
     - (e_1 … e_n) is evaluated as a ***call of the function given by*** e_1; the values obtained by evaluating e_2, ..., e_n are passed to the call as argument values.
     - **Example**: (+ e_2 e_3) is evaluated as a ***call of*** +; the argument values are obtained by evaluating e_2 and e_3.
   - If e_1 is ***the name of a special operator or macro***, then:
     -
     -

How S-Expressions are Evaluated

1. A **symbol** evaluates to the value of the variable / formal parameter or constant that is denoted by the symbol.
2. Any **atom that is not a symbol** evaluates to the same atom.
3. A **nonempty proper list** (e_1 … e_n) may be evaluated as a ***function call*** ***or*** as a ***special/macro form*** depending on e_1, the ***first element*** of the list:
   - If e_1 is ***the name of a function*** or e_1 is ***a lambda expression*** (a kind of list we will study later), then:
     - (e_1 … e_n) is evaluated as a ***call of the function given by*** e_1; the values obtained by evaluating e_2, ..., e_n are passed to the call as argument values.
     - **Example**: (+ e_2 e_3) is evaluated as a ***call of*** +; the argument values are obtained by evaluating e_2 and e_3.
   - If e_1 is ***the name of a special operator or macro***, then:
     - (e_1 … e_n) is evaluated as a ***special/macro form*** in a way that depends on the special operator/macro.
     - **Example**:

How S-Expressions are Evaluated

1. A **symbol** evaluates to the value of the variable / formal parameter or constant that is denoted by the symbol.
2. Any **atom that is not a symbol** evaluates to the same atom.
3. A **nonempty proper list** (e_1 … e_n) may be evaluated as a ***function call*** ***or*** as a ***special/macro form*** depending on e_1, the ***first element*** of the list:
   - If e_1 is ***the name of a function*** or e_1 is ***a lambda expression*** (a kind of list we will study later), then:
     - (e_1 … e_n) is evaluated as a ***call of the function given by*** e_1; the values obtained by evaluating e_2, ..., e_n are passed to the call as argument values.
     - **Example**: (+ e_1 e_2) is evaluated as a ***call of*** +; the argument values are obtained by evaluating e_1 and e_2.
   - If e_1 is ***the name of a special operator or macro***, then:
     - (e_1 … e_n) is evaluated as a ***special/macro form*** in a way that depends on the special operator/macro.
     - **Example**: (SETF X 9) is evaluated as a ***macro form***.

How S-Expressions are Evaluated

3. A **nonempty proper list** (e_1 … e_n) may be evaluated as a ***function call*** ***or*** as a ***special/macro form*** depending on e_1, the ***first element*** of the list:
   - If e_1 is ***the name of a function*** or e_1 is ***a lambda expression*** (a kind of list we will study later), then:
     - (e_1 … e_n) is evaluated as a ***call of the function given by*** e_1; the values obtained by evaluating e_2, ..., e_n are passed to the call as argument values.

   - If e_1 is ***the name of a special operator or macro***, then:
     - (e_1 … e_n) is evaluated as a ***special/macro form*** in a way that depends on the special operator/macro.

How S-Expressions are Evaluated

3. A **nonempty proper list** (e_1 … e_n) may be evaluated as a ***function call*** ***or*** as a ***special/macro form*** depending on e_1, the ***first element*** of the list:
   - If e_1 is ***the name of a function*** or e_1 is ***a lambda expression*** (a kind of list we will study later), then:
     - (e_1 … e_n) is evaluated as a ***call of the function given by*** e_1; the values obtained by evaluating e_2, ..., e_n are passed to the call as argument values.
   - If e_1 is ***the name of a special operator or macro***, then:
     - (e_1 … e_n) is evaluated as a ***special/macro form*** in a way that depends on the special operator/macro.

How S-Expressions are Evaluated

3. A **nonempty proper list** (e_1 … e_n) may be evaluated as a ***function call*** ***or*** as a ***special/macro form*** depending on e_1, the ***first element*** of the list:
   - If e_1 is ***the name of a function*** or e_1 is ***a lambda expression*** (a kind of list we will study later), then:
     - (e_1 … e_n) is evaluated as a ***call of the function given by*** e_1; the values obtained by evaluating e_2, ..., e_n are passed to the call as argument values.
   - If e_1 is ***the name of a special operator or macro***, then:
     - (e_1 … e_n) is evaluated as a ***special/macro form*** in a way that depends on the special operator/macro.

A list (e_1 … e_n) can be evaluated ***only*** in the above cases:

-

Note:

How S-Expressions are Evaluated

3. A **nonempty proper list** (e_1 … e_n) may be evaluated as a ***function call*** ***or*** as a ***special/macro form*** depending on e_1, the ***first element*** of the list:
   - If e_1 is ***the name of a function*** or e_1 is ***a lambda expression*** (a kind of list we will study later), then:
     - (e_1 … e_n) is evaluated as a ***call of the function given by*** e_1; the values obtained by evaluating e_2, ..., e_n are passed to the call as argument values.
   - If e_1 is ***the name of a special operator or macro***, then:
     - (e_1 … e_n) is evaluated as a ***special/macro form*** in a way that depends on the special operator/macro.

A list (e_1 … e_n) can be evaluated ***only*** in the above cases:

- If e_1 is ***neither*** a **symbol that is the name of a function** ***nor*** a **symbol that is the name of a special operator/macro** ***nor*** a **lambda expression**, then an ***error*** will occur when (e_1 … e_n) is evaluated!

Note:

How S-Expressions are Evaluated

3. A **nonempty proper list** (e_1 … e_n) may be evaluated as a ***function call*** ***or*** as a ***special/macro form*** depending on e_1, the ***first element*** of the list:
   - If e_1 is ***the name of a function*** or e_1 is ***a lambda expression*** (a kind of list we will study later), then:
     - (e_1 … e_n) is evaluated as a ***call of the function given by*** e_1; the values obtained by evaluating e_2, ..., e_n are passed to the call as argument values.
   - If e_1 is ***the name of a special operator or macro***, then:
     - (e_1 … e_n) is evaluated as a ***special/macro form*** in a way that depends on the special operator/macro.

A list (e_1 … e_n) can be evaluated ***only*** in the above cases:

- If e_1 is ***neither*** a **symbol that is the name of a function** ***nor*** a **symbol that is the name of a special operator/macro** ***nor*** a **lambda expression**, then an ***error*** will occur when (e_1 … e_n) is evaluated!

Note: The result of evaluating a *dotted* list is undefined!