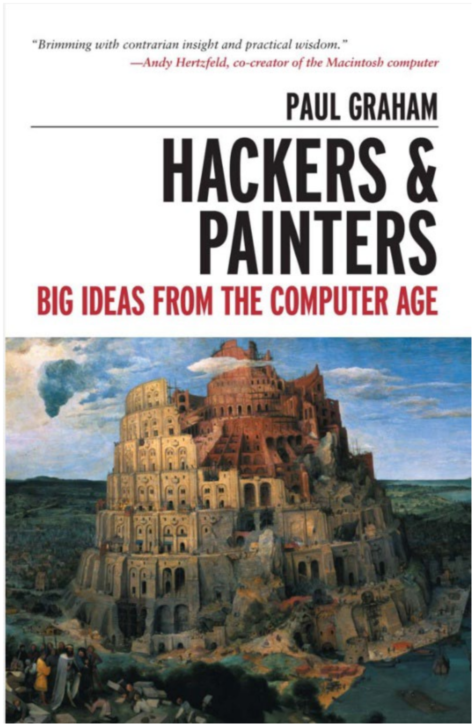
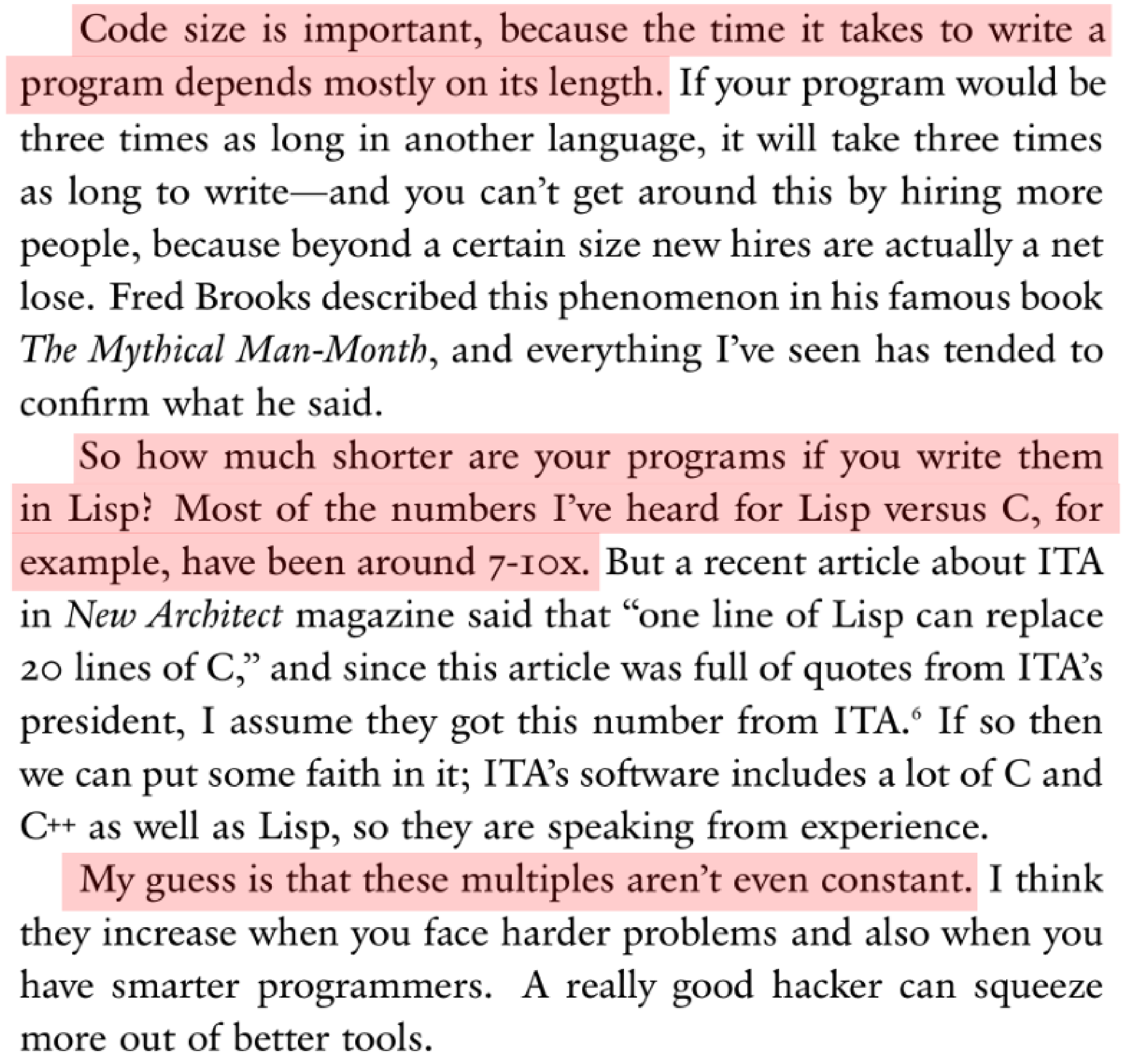
- Many Lisp programmers consider Lisp to be a language in which an able programmer or a small team of able programmers can be particularly productive.
- One reason that has been given for this is that Lisp code is often more *succinct* than code that solves the same problem in more widely used languages. (Lisp macros can be used to reduce the code size of medium and large projects in ways that would not be possible in most other languages.)

  An excerpt from the (fairly well known) book
  Paul Graham, *Hackers & Painters*, O'Reilly, 2004
  that makes this argument is given below.


"Brimming with contrarian insight and practical wisdom."
—Andy Hertzfeld, co-creator of the Macintosh computer
PAUL GRAHAM
HACKERS & PAINTERS
BIG IDEAS FROM THE COMPUTER AGE

From: Paul Graham, *Hackers & Painters*, O'Reilly, 2004, pp. 192–93.



Code size is important, because the time it takes to write a program depends mostly on its length. If your program would be three times as long in another language, it will take three times as long to write—and you can't get around this by hiring more people, because beyond a certain size new hires are actually a net lose. Fred Brooks described this phenomenon in his famous book *The Mythical Man-Month*, and everything I've seen has tended to confirm what he said.

So how much shorter are your programs if you write them in Lisp? Most of the numbers I've heard for Lisp versus C, for example, have been around 7-10x. But a recent article about ITA in *New Architect* magazine said that "one line of Lisp can replace 20 lines of C," and since this article was full of quotes from ITA's president, I assume they got this number from ITA.[6] If so then we can put some faith in it; ITA's software includes a lot of C and C++ as well as Lisp, so they are speaking from experience.

My guess is that these multiples aren't even constant. I think they increase when you face harder problems and also when you have smarter programmers. A really good hacker can squeeze more out of better tools.

- Lisp has long had some extremely enthusiastic advocates (including influential computer scientists and programmers), certain of whom have even claimed that Lisp is the best programming language.
- A few of the quotes on Lisp's Wikiquote page https://en.wikiquote.org/wiki/Lisp_(programming_language) illustrate this.
- The following interesting article considers the question of how Lisp came to have such devoted fans:

  How Lisp Became God's Own Programming Language (S. Target, 2018)

You will use Lisp for pure functional programming. (More specifically, you will use Common Lisp, which is one of the principal dialects of Lisp.)

However, Lisp does ***not*** constrain programmers to programming functionally! Indeed, Lisp provides the following, which are ***not*** used in pure functional programming:

- Assignment forms (e.g., SETF forms in Common Lisp) that update values of variables and data structure components.
- Iterative (looping) constructs (e.g., DO in Common Lisp).
- Functions that perform I/O (e.g., FORMAT in Common Lisp).

Warning: When doing assignments and answering exam questions, you ***must not*** write Lisp code that uses any of the above unless you are told you may do so!

  - **Note**: Much of the code in ch. 9 (Input/Output), ch. 10 (Assignment), and ch. 11 (Iteration and Block Structure) of Touretzky uses the above features and would be ***un***acceptable unless otherwise indicated!

The term Lisp is often used to mean the Common Lisp dialect of Lisp. Other important dialects of Lisp are:

Scheme This Lisp dialect is used in the course reader.

Important: Exam questions may require you to ***read*** simple Scheme code, but won't expect you to write Scheme code. Solutions to Lisp Assignments 3 – 5 will be provided in Scheme.

Racket A dialect of Scheme developed from the mid-90s onwards; it was called PLT Scheme until 2010.

Clojure A relatively(!) young Lisp dialect (its 1st version was released in 2007) that compiles to Java bytecodes and can use Java classes.

- Some other languages that offer excellent or at least good support for functional programming are: Haskell, OCaml, SML, F#, Erlang, Scala, Kotlin, Javascript, and R.

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".

- 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)]

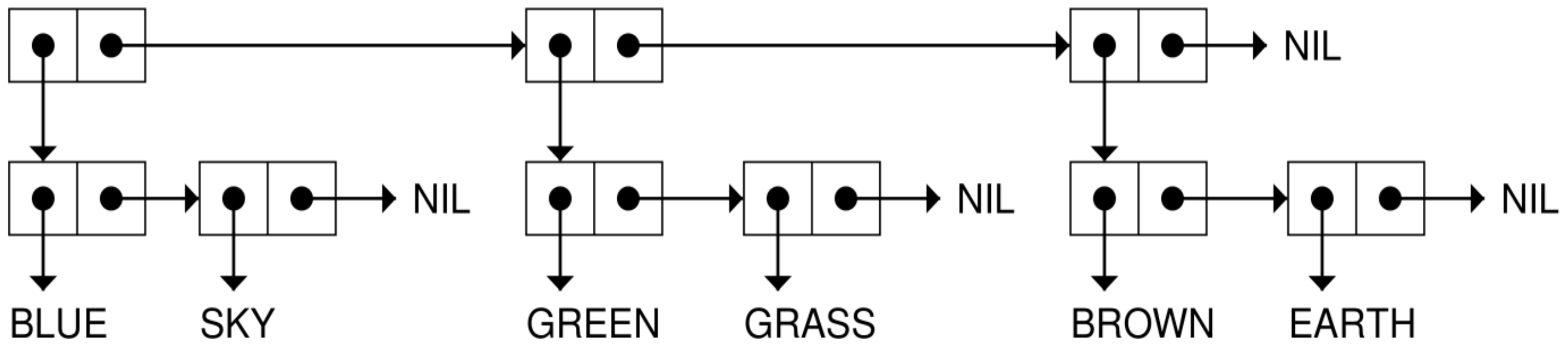
  If a function you write for this course returns a dotted list, then either your code has a bug or an inappropriate argument value was passed to the function.

- Each S-expression is a textual representation of a Lisp data object that's *also* called an S-expression.
- 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. 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 10 23))`
   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,

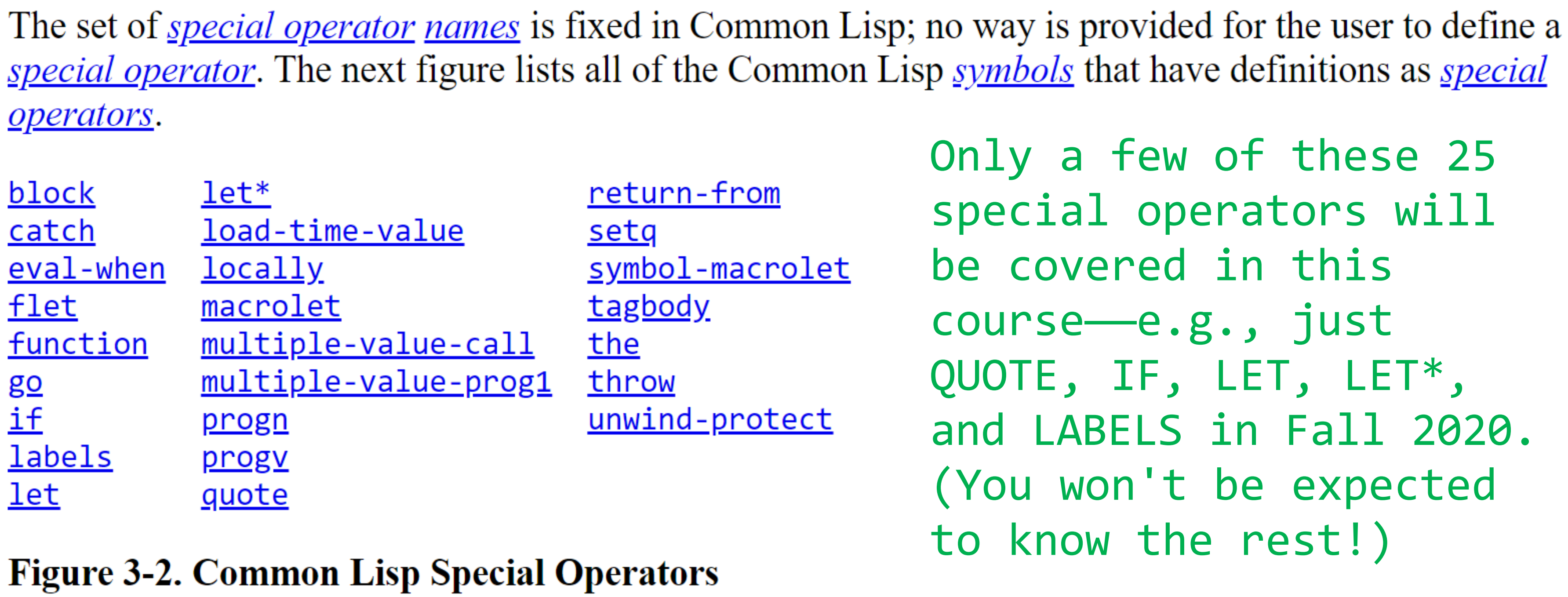
   ```
           (sqrt (+ (* 3 2) (- 4 1)))
   ```

   is an S-expression that can be evaluated.
2. As Lisp **data**. Example: `((john smith) (2001 10 23))`

- 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.
  - Ch. 14 of Touretzky explains how to write macros, but you will ***<u>not</u>*** be expected to do that: All the macros you need to use will be predefined (i.e., built-in) macros.

- Many commonly used Lisp "keywords" (e.g., SETF, DEFUN, COND, AND, OR) are predefined macros that are or could be defined in terms of built-in Lisp functions (e.g., CAR, CDR) and 25 built-in keywords called *special operators*:

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

Only a few of these 25 special operators will be covered in this course—e.g., just QUOTE, IF, LET, LET*, and LABELS in Fall 2020. (You won't be expected to know the rest!)

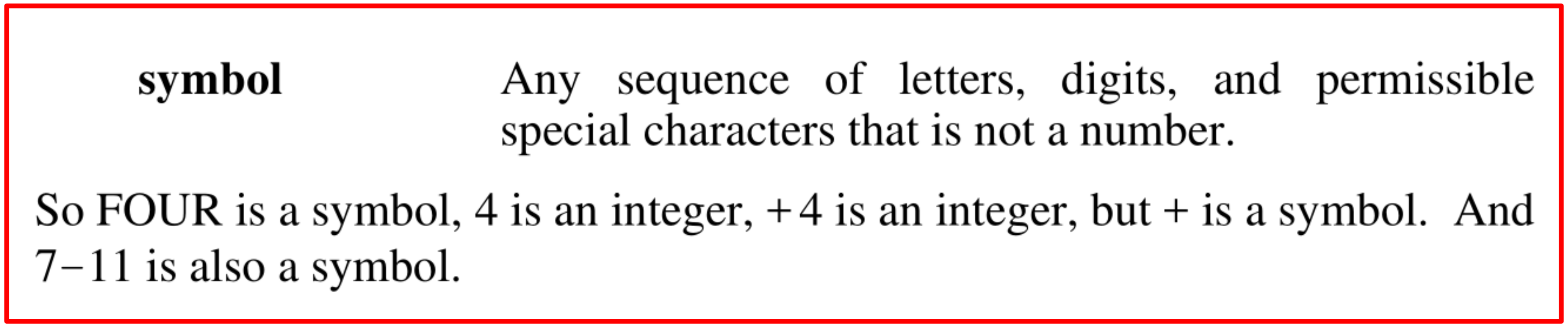
More on Symbols and Lists in Common Lisp

- 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:

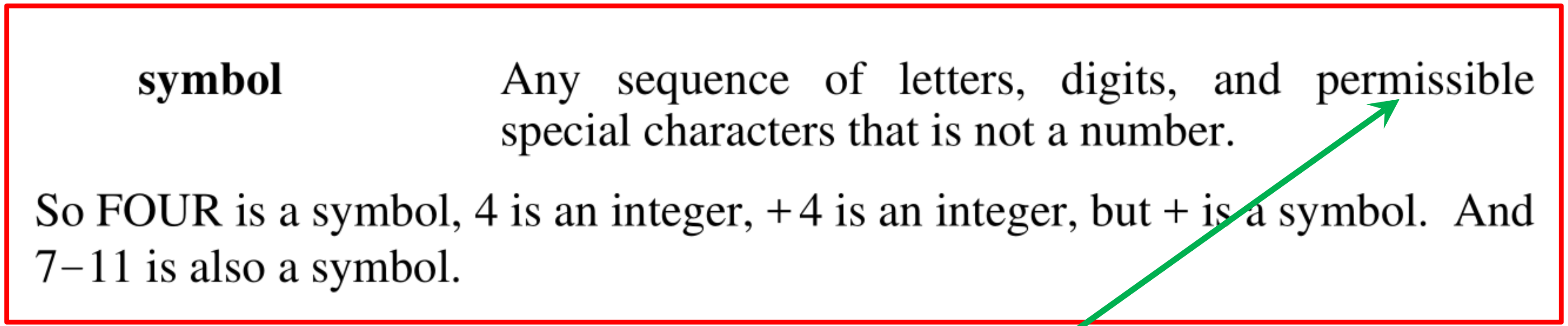


symbol Any sequence of letters, digits, and permissible special characters that is not a number.

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:



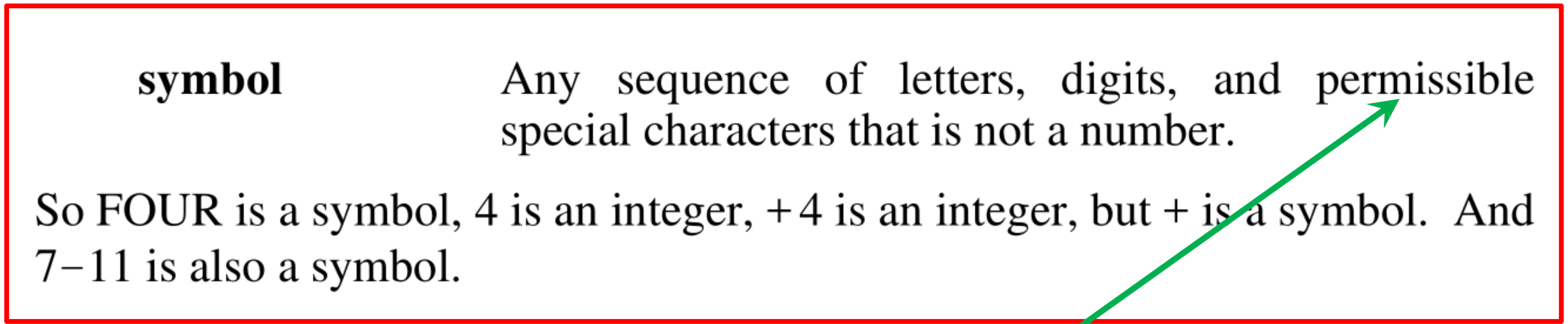
symbol Any sequence of letters, digits, and permissible special characters that is not a number.

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"?

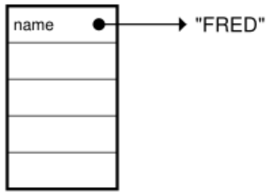
A. Any character that is ***not*** a whitespace character and also is ***not*** one of ( ) ' ` " , ; : | \ is permissible, but # can't be the *first* character of a symbol and a symbol can't consist only of . characters.

Symbols 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.

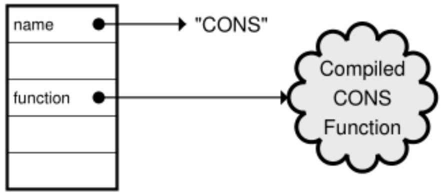
3.18 INTERNAL STRUCTURE OF SYMBOLS

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:

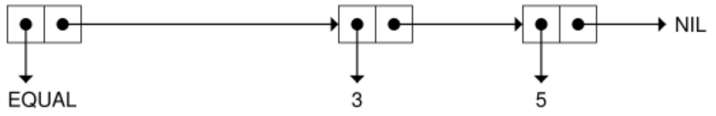


The ''FRED'' appearing above in quotation marks is called a **string**. Strings are sequences of characters; they will be covered more fully in Chapter 9. For now it suffices to note that strings are used to store the names of symbols; a symbol and its name are actually two different things.

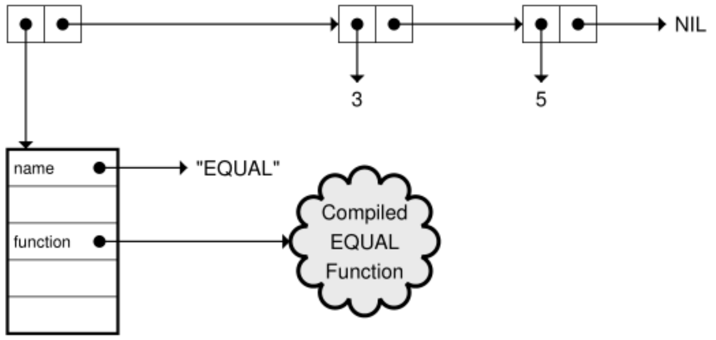
Some symbols, like CONS or +, are used to name built-in Lisp functions. The symbol CONS has a pointer in its **function cell** to a ''compiled code object'' that represents the machine language instructions for creating new cons cells.



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 can be used in Lisp as follows:

- as names of ***variables/parameters*** and *constants*.
- as names of (ordinary) ***functions*** (e.g., +, −, SQRT).
- as names of ***special operators*** (also called *special functions*) and ***macros*** (also called *macro functions*).
  - 4 *special operators* that will be used many times in this course are IF, QUOTE, LET, and LET*.
  - 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 have been or 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 can be used in Lisp as follows:

- as names of ***variables/parameters*** and *constants*.
- as names of (ordinary) ***functions*** (e.g., +, −, SQRT).
- as names of ***special operators*** (also called *special functions*) and ***macros*** (also called *macro functions*).
  - 4 *special operators* that will be used many times in this course are IF, QUOTE, LET, and LET*.
  - 6 predefined macros that have been or 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 can also be 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?
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.

Lists

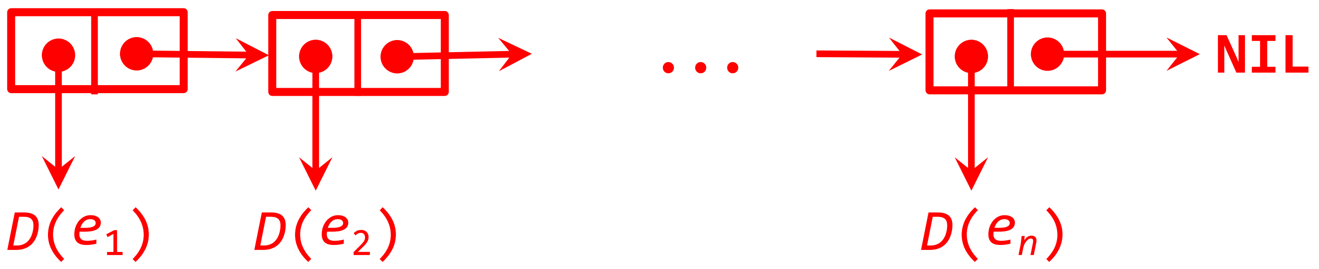
As mentioned earlier, there are two kinds of list:
(1) proper lists (2) dotted lists

- *Proper* lists of length 0, 1, 2, ..., n have the forms
  () or NIL, (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:

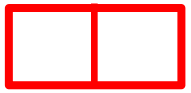
Lists

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
  () or NIL, (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:

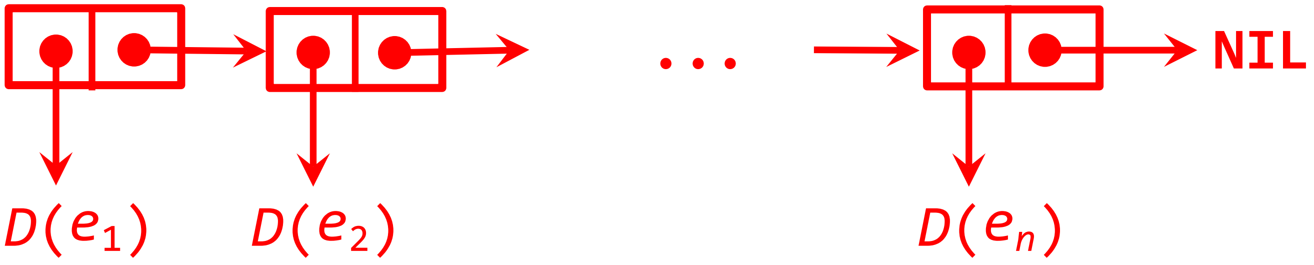


Here $D(e_i)$ is a drawing of the structure e_i represents.

Each  box depicts a ***cons cell***; this has ***car*** and ***cdr*** fields (depicted by the left and right parts of the box).

Lists

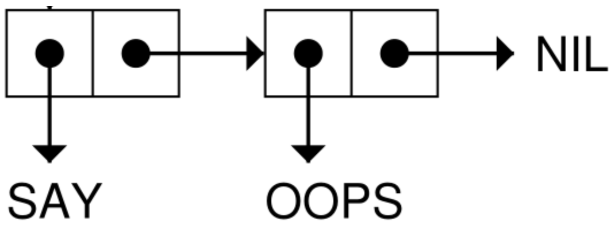
- A *proper* list (e_1 … e_n) represents a singly-linked list data structure that may be drawn as follows:



Here $D(e_i)$ is a drawing of the structure e_i represents.

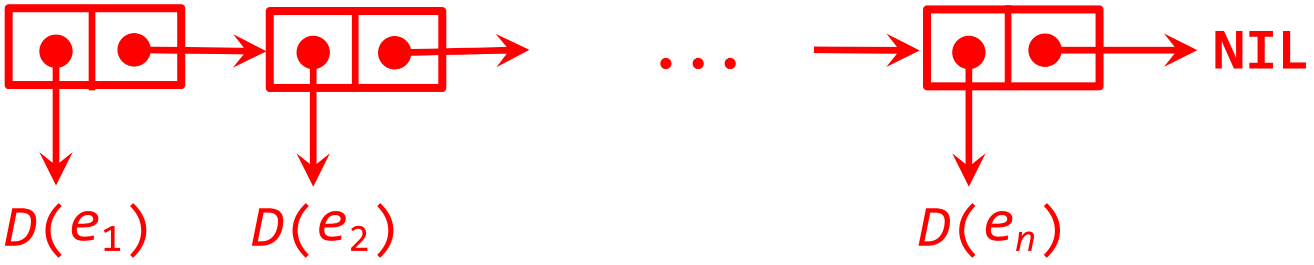
Example from p. 34 of Touretzky.

(SAY OOPS) represents:



Lists

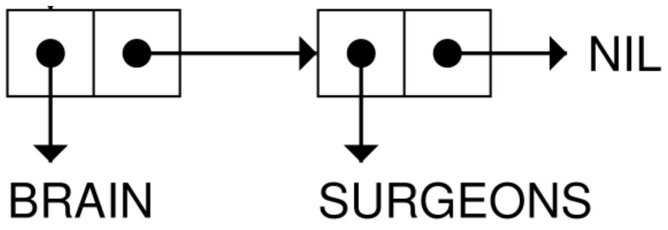
- A *proper* list (e_1 … e_n) represents a singly-linked list data structure that may be drawn as follows:



Here $D(e_i)$ is a drawing of the structure e_i represents.

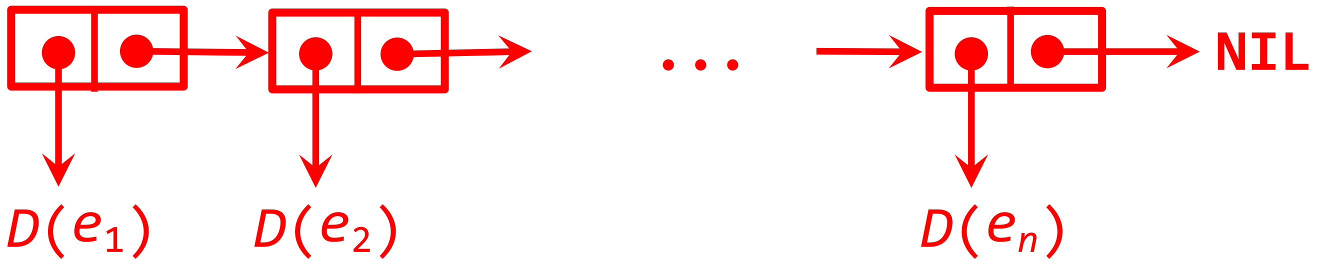
From p. 34 of Touretzky.

(BRAIN SURGEONS) represents:



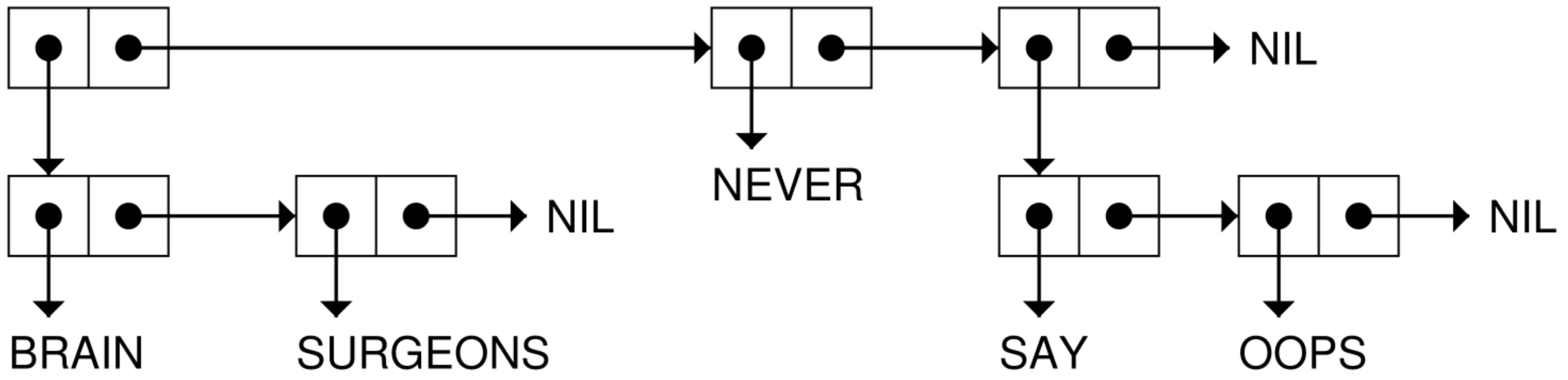
Lists

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



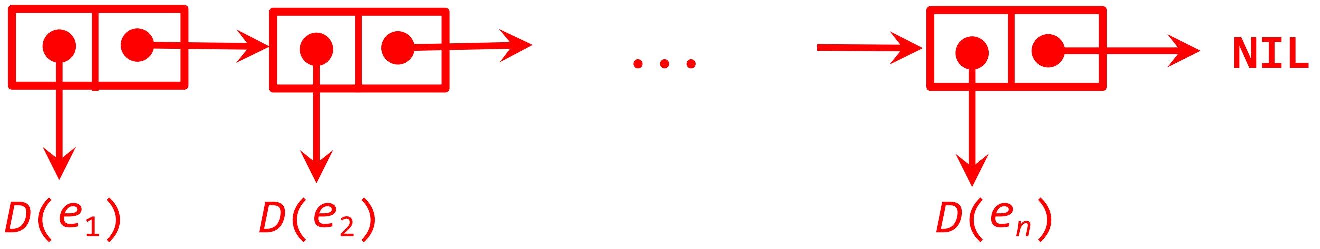
Here $D(e_i)$ is a drawing of the structure e_i represents.

From p. 34 of Touretzky.
((BRAIN SURGEONS) NEVER (SAY OOPS)) represents:



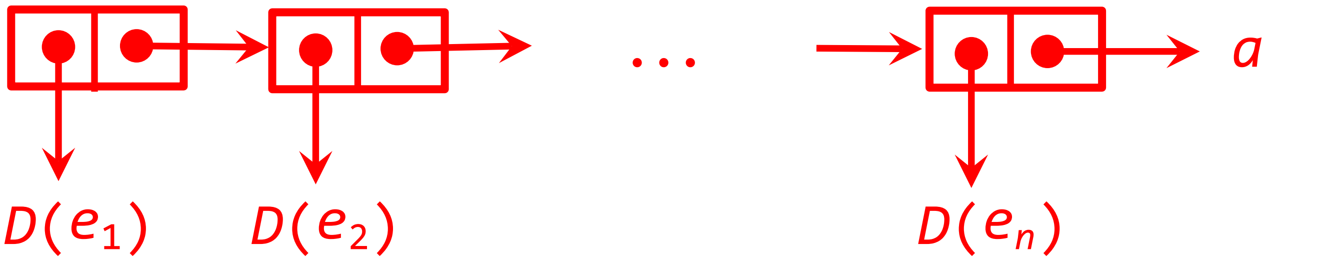
Lists

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



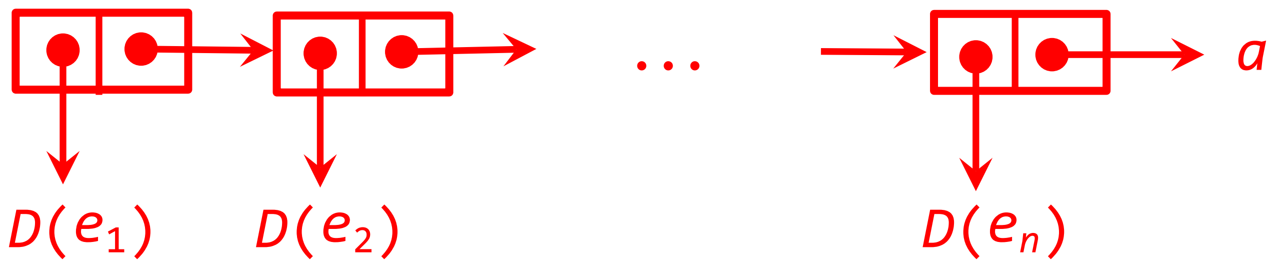
Here $D(e_i)$ is a drawing of the structure e_i represents.

- 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**.
- If e is a list, then (e_1 … e_n . e) is a legal S-expression but need not be a dotted list: We'll see that it is a proper list if e is.
- If a is an atom, then (e_1 … e_n . a) represents a singly-linked list data structure that may be drawn as follows:



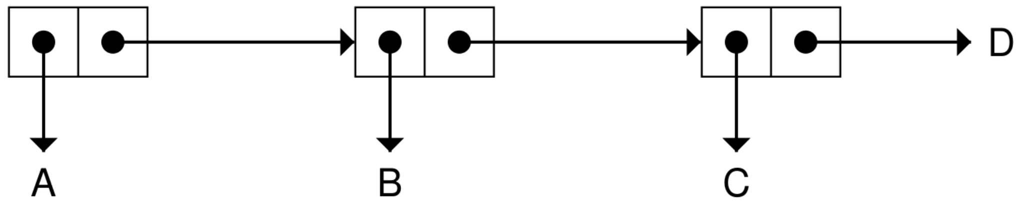
Lists

- 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**.
- If e is a list, then (e_1 … e_n . e) is a legal S-expression but need not be a dotted list: We'll see that it is a proper list if e is.
- If a is an atom, then (e_1 … e_n . a) represents a singly-linked list data structure that may be drawn as follows:



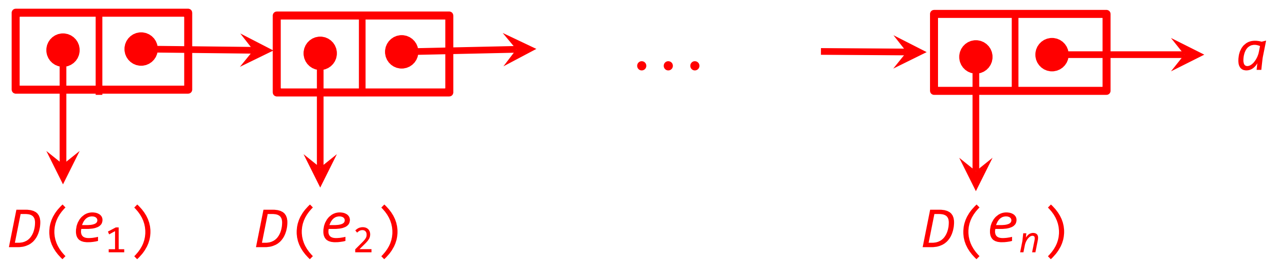
Examples from pp. 72-3 of Touretzky

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



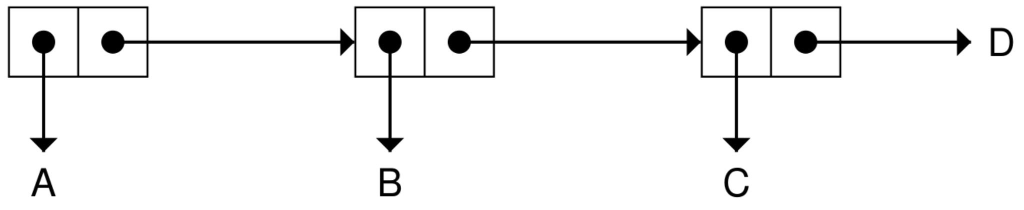
Lists

- 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**.
- If e is a list, then (e_1 … e_n . e) is a legal S-expression but need not be a dotted list: We'll see that it is a proper list if e is.
- 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

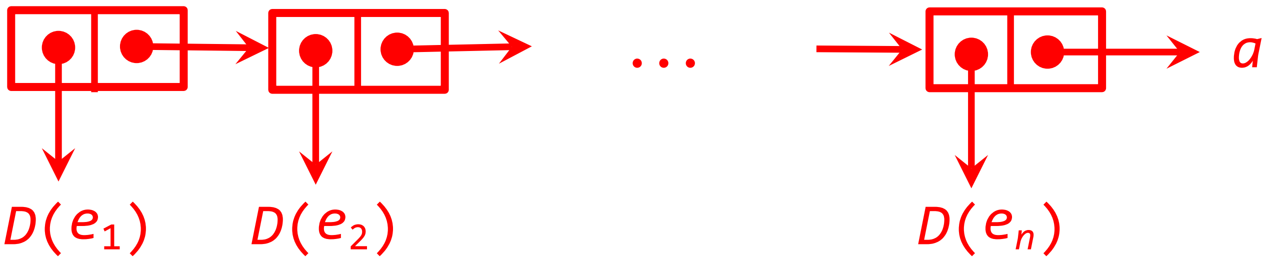
The dotted list (A B C . D) represents:



The dotted pair (A . B) represents:

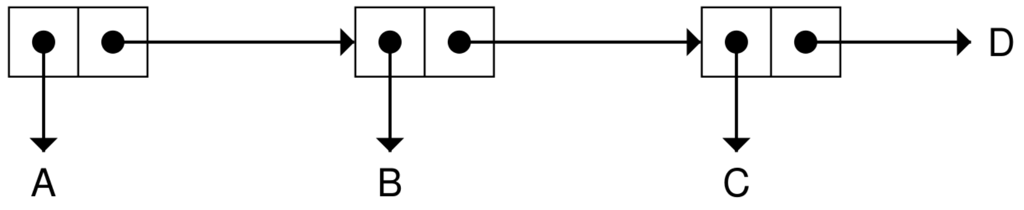
Lists

- 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**.
- If e is a list, then (e_1 … e_n . e) is a legal S-expression but need not be a dotted list: We'll see that it is a proper list if e is.
- If a is an atom, then (e_1 … e_n . a) represents a singly-linked list data structure that may be drawn as follows:

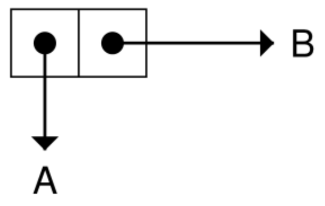


Examples from pp. 72-3 of Touretzky

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

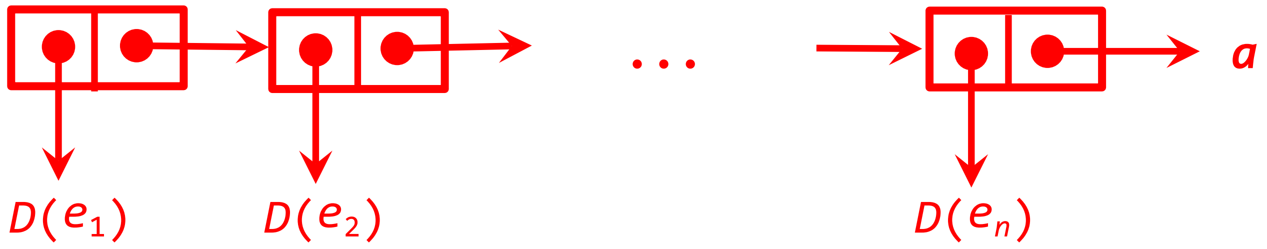


The dotted pair (A . B) represents:

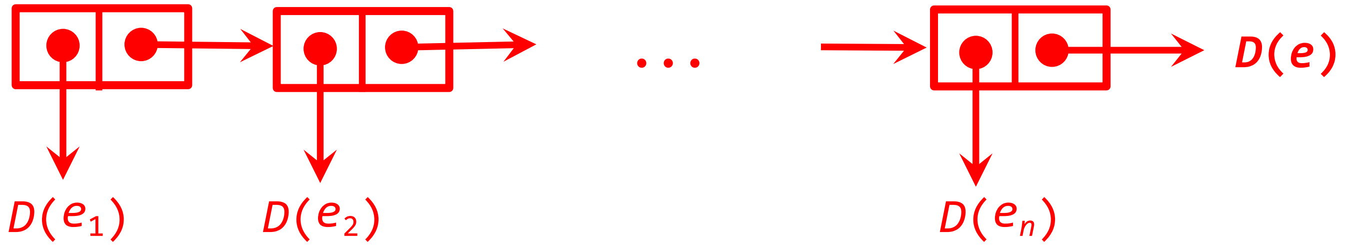


Lists

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

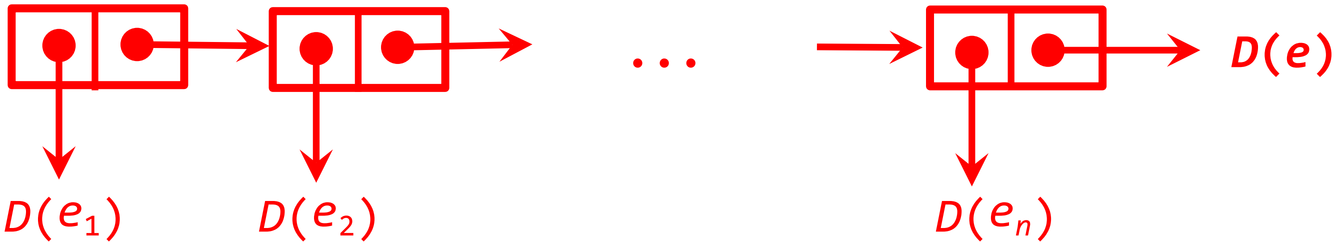


- More generally, if e is any S-expression, then (e_1 … e_n . e) represents a data structure that may be drawn as follows:

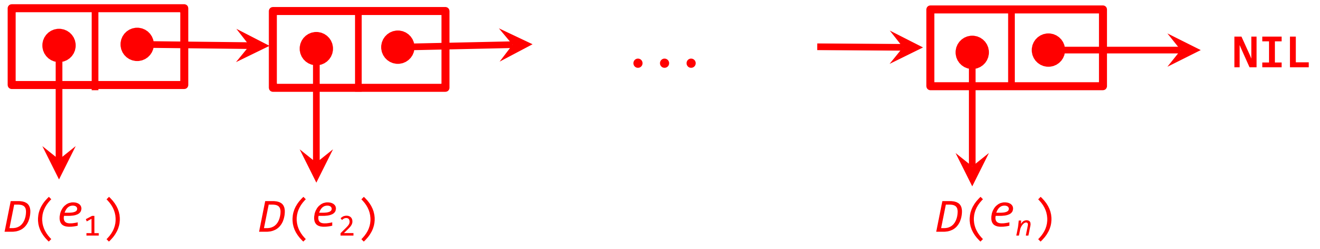


Lists

- If e is any S-expression, then $(e_1 \ldots e_n \;.\; e)$ represents a data structure that may be drawn as follows:



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



- Therefore $(e_1 \ldots e_n \;.\; e)$ can be simplified if e is a list:

$(e_1 \ldots e_n$. 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 usually write $(e_1 \ldots e_n \;.\; e)$ ***<u>only if e is an atom other than</u>*** <u>NIL</u>.