

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:

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	prog1	
let	quote	

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!

Figure 3-2. Common Lisp Special Operators

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 -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:

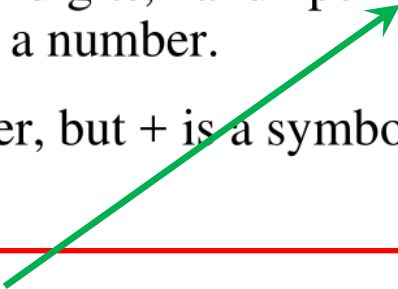
- Integers (e.g., 3 or -8468284739390847398474784894)
 - Ratios (e.g., -41/31) were described [earlier](#).
 - Floating point numbers (e.g., 12.876 or 2.31e-3)
 - Complex numbers (e.g., #C(3 -5/8) or #C(2.3 7.2))
 - #C(x y) represents $x + yi$, where $i = \sqrt{-1}$.
 - We will **not** use complex numbers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



Q. Which special characters are "permissible"?

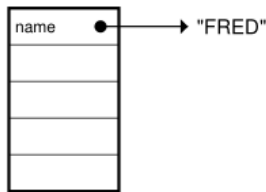
A. Any character that is not a whitespace character and also is not one of () ' ` " , ; : | \ is permissible, but # can't be the *first* character of a symbol and a symbol can't consist only of . characters.

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.

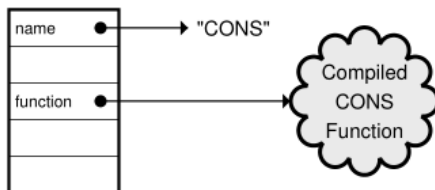
3.18 INTERNAL STRUCTURE OF SYMBOLS

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:

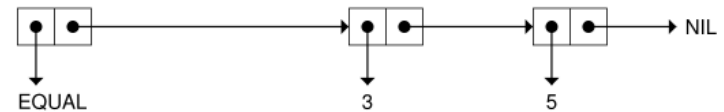


The "FRED" appearing above in quotation marks is called a **string**. Strings are sequences of characters; they will be covered more fully in Chapter 9. For now it suffices to note that strings are used to store the names of symbols; a symbol and its name are actually two different things.

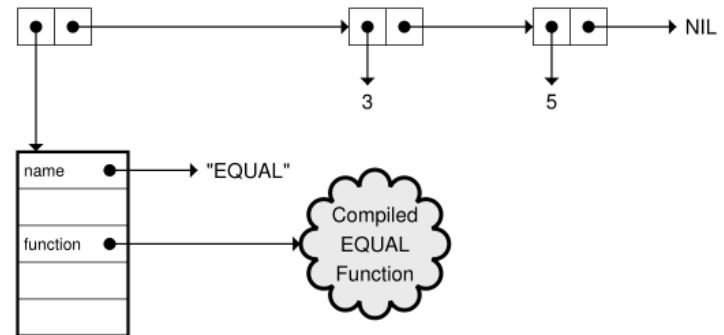
Some symbols, like CONS or +, are used to name built-in Lisp functions. The symbol CONS has a pointer in its **function cell** to a "compiled code object" that represents the machine language instructions for creating new cons cells.



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:

- 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 be 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 be 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?

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

- **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

- *Proper* lists of length $0, 1, 2, \dots, n$ 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.
- A *proper* list $(e_1 \dots e_n)$ 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

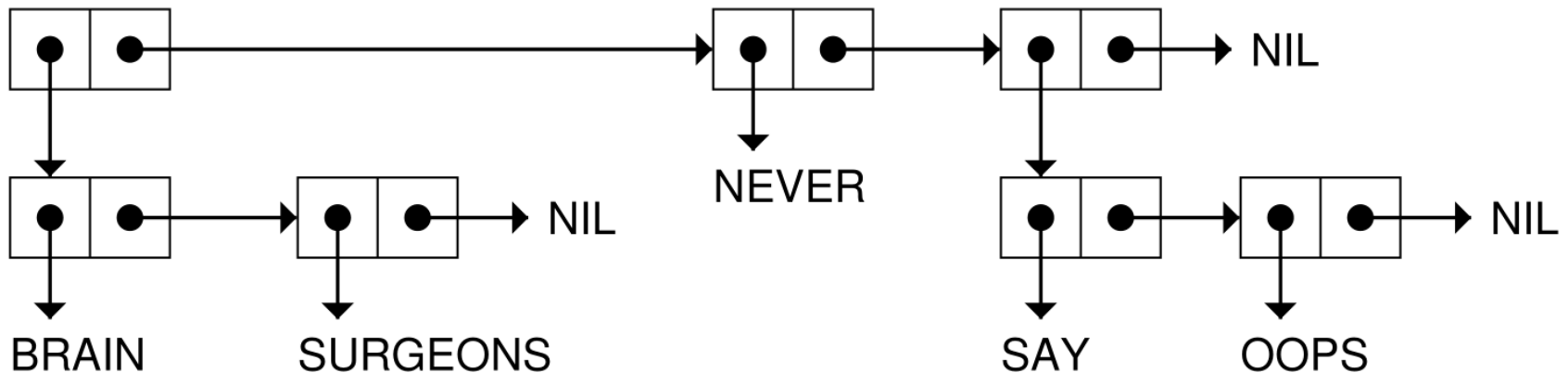
- A *proper* list $(e_1 \dots e_n)$ represents a singly-linked list data structure that may be drawn as follows:



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

From p. 34 of Touretzky.

$((\text{BRAIN SURGEONS}) \text{ NEVER } (\text{SAY OOPS}))$ represents:



Lists

- A *proper* list $(e_1 \dots e_n)$ 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 $(e_1 \dots e_n . a)$ where $n \geq 1$, each e is an S-expression, and a is an atom other than NIL.
- The notation $(e_1 \dots e_n . a)$ can also be used if a is a list.
- 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:



Lists

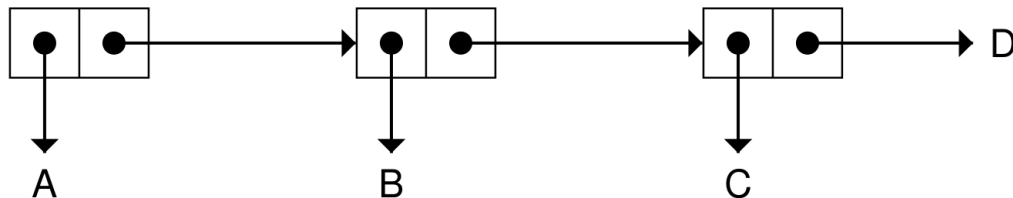
- A *dotted* list has the form $(e_1 \dots e_n . a)$ where $n \geq 1$, each e is an S-expression, and a is an atom other than NIL.
- The notation $(e_1 \dots e_n . a)$ can also be used if a is a list.
- If a is an atom, then $(e_1 \dots e_n . a)$ represents a singly-linked list data structure that may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



Examples from pp. 72-3 of Touretzky

The dotted list

(A B C . D) represents:



The dotted pair

(A . B) represents:

