Java is designed for object-oriented imperative programming, and is not ideally suited to functional programming.

For example, here are two features Java lacks that we will want to make use of in functional programming:

- Java has no predefined singly-linked list class.

  Singly-linked lists are very commonly used in functional programming.

  This simple and versatile kind of data structure is well suited to functional programming, since ***a list* L1 *of this kind is <u>unchanged</u> when we create a new list* L2 *from it by adding one or more nodes at the front***:

  Many languages designed to support functional programming have a predefined singly-linked list class, and some predefined functions that perform common operations on such lists to create new lists ***<u>without changing existing lists</u>***.

Java is designed for object-oriented imperative programming, and is not ideally suited to functional programming.

For example, here are two features Java lacks that we will want to make use of in functional programming:

- Java has no predefined singly-linked list class.

  Many languages designed to support functional programming have a predefined singly-linked list class, and some predefined functions that perform common operations on such lists to create new lists ***without changing existing lists***.

- Java does ***not*** support tail-recursion optimization.

  A kind of recursion called ***tail recursion*** is quite often used in functional programming—see, e.g., secs. 8.11 and 8.16 of Touretzky. If the compiler or interpreter performs ***tail recursion optimization***, tail recursion is executed very efficiently and there's no limit on the depth of tail recursion. But Java compilers ***don't***: Tail recursion in Java is no more efficient than other recursion, and stack overflow occurs when the depth of recursion is too great.

It is currently uncommon to use Java in a purely functional style.

But it’s common in Java to use *higher-order functions** (especially functions that take functions as arguments) and to use *lambda expressions* to create functions--both of these are important aspects of functional programming.

*A ***higher-order function*** is a function that ***either*** takes a function as argument ***or*** returns a function as its result.

Example The following Java code uses a lambda expression and two functions that take functions as arguments:

```
Stream<String> s = Stream.of("Our", "course", "is", "CSCI 316", "!");
s.map(String::length).forEach(x -> System.out.print(" " + x));
```

prints: **3 6 2 8 1**   [**map** & **forEach** are higher-order functions.]

This ***isn't*** an example of pure functional programming, for 2 reasons: The code performs output, and the Stream referenced by **s** can't be used again after it executes.

It is currently uncommon to use Java in a purely functional style.

The language we will use, Lisp, is better-suited to functional programming.

- The first version of Lisp was designed by McCarthy at MIT in 1958, which makes Lisp the 2nd oldest high-level language that is still in use (after Fortran).

- McCarthy was one of the founders of the field of Artificial Intelligence (AI); he won the 1971 Turing Award for his contributions to that field. For 25+ years (mid-60s – early 90s) Lisp was the dominant programming language in the AI research community.

- "Lisp" is an acronym for LISt Processor: Lisp's fundamental data structure is the list (e_1 e_2 ... e_n).

  Here ***any of the*** e***'s may also be a list***; arbitrarily complex data structures can be implemented as lists.

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(!) new 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, SML, F#, OCaml, Scala, Erlang, Javascript, and Kotlin.

Getting Started with the CLISP Common Lisp Interpreter

- While logged in to **euclid** or **venus**, enter

  cl

  at the shell prompt to start the CLISP interpreter.
- CLISP's prompt is **[*n*]>** (e.g., **[1]>**, **[2]>**, **[3]>**, ...), where *n* is a count of the number of prompts that have been displayed so far.
- At any prompt, you can exit from CLISP by typing CTRL d or by entering (exit) [*including* the parentheses!].
- At any prompt, you can enter a Lisp expression to be evaluated. Lisp will ***read*** in your expression, ***evaluate*** it, and then ***print*** the expression's value.
- Lisp expressions are written in a special notation:

Java: 3 - 4.1	Lisp: (- 3 4.1)
Java: 3 - 4.1 + 2	Lisp: (+ (- 3 4.1) 2)
Java: Math.sqrt(3*4.1)	Lisp: (sqrt (* 3 4.1))

From p. 3 of Touretzky:

In this book we will work mostly with **integers**, which are whole numbers. Common Lisp provides many other kinds of numbers. One kind you should know about is **floating point** numbers. A floating point number is always written with a decimal point; for example, the number five would be written 5.0. The SQRT function generally returns a floating point number as its result, even when its input is an integer.

Note: In Clisp, SQRT returns an integer if its argument is a perfect square.

$$\begin{tikzcd} 25 \arrow[r] & \boxed{\text{SQRT}} \arrow[r] & \cancel{5.0}\ \text{5 in Clisp} \end{tikzcd}$$

Ratios are yet another kind of number. On a pocket calculator, one-half must be written in floating point notation, as 0.5, but in Common Lisp we can also write one-half as the ratio 1/2. Common Lisp automatically simplifies ratios to use the smallest possible denominator; for example, the ratios 4/6, 6/9, and 10/15 would all be simplified to 2/3.

Getting Started with the CLISP Common Lisp Interpreter

- At any prompt, you can enter a Lisp expression to be evaluated. Lisp will ***read*** in your expression, ***evaluate*** it, and then ***print*** the expression's value.
- Lisp expressions are written in a special notation:

```
Java: 3 – 4.1                    Lisp: (– 3 4.1)
Java: 3 – 4.1 + 2                Lisp: (+ (– 3 4.1) 2)
Java: Math.sqrt(3*4.1)           Lisp: (sqrt (* 3 4.1))
```

  - If an integer that is a perfect square (e.g., 9) is passed as argument to CLISP's sqrt, its ***integer*** square root is returned; in some other implementations of Common Lisp, a floating point result is returned.
- If evaluation of an expression produces an error, then CLISP prints an error message followed by a **Break ... >** prompt: You can enter **:q** at a **Break ... >** prompt to get back to the regular [*n*]> prompt!

  Example: The function sqrt expects just one argument, so evaluation of (sqrt 4 5) produces a **Break ... >**.

Assigning Values to Global Variables

- **SETF** can be used to assign a value to a variable: The value of the Lisp expression (setf x *expr*) is the value of *expr*--e.g., the value of (+ (setf x 3) 5) is 8--but evaluation of (setf x *expr*) has the side-effect of assigning *expr*'s value to the variable x.
- Thus (setf x ...) is analogous to a C++ or Java expression of the form (x = ...)!
- Once a variable has been assigned a value, the variable can be used to represent that value in subsequent expressions.
- **IMPORTANT**: SETF is ***not*** used in pure functional programming, so the Lisp functions you write when doing programming assignments or answering exam questions must ***not*** use SETF!

Assigning Values to Global Variables

- **SETF** can be used to assign a value to a variable: The value of the Lisp expression (setf x *expr*) is the value of *expr*--e.g., the value of (+ (setf x 3) 5) is 8--but evaluation of (setf x *expr*) has the side-effect of assigning *expr*'s value to the variable x.

- **IMPORTANT**: SETF is ***not*** used in pure functional programming, so the Lisp functions you write when doing programming assignments or answering exam questions must ***not*** use SETF!

- You may use SETF when ***testing*** your functions:

  For example, if you plan to use $2^{31} - 1$ as a test argument value several times, then you can use SETF to store $2^{31} - 1$ in a variable that will be used as the actual argument each time.

Ratios (a type of number that represents fractions)

- Unlike C++ and Java, Common Lisp has a built-in data type called ***ratio*** that represents (positive and negative, proper and improper) fractions ***exactly***, with no rounding error. Examples:

  6/8 represents the positive proper fraction $\frac{6}{8} = \frac{3}{4}$.

  -5/4 represents the negative improper fraction $\frac{-5}{4}$.

  - Ratios are always ***printed in lowest terms*** (but they need not be written in lowest terms): You can write 6/8, but this ratio would be printed as 3/4.
  - If m and n are integers, n is not 0, and n does not divide m, then the value of (/ m n) is a ratio. For example, the value of (/ 12 9) is 12/9 = 4/3.
  - There is no space before or after the / in a ratio: 5/7 ***cannot*** be written as 5 /7 or 5/ 7.

Ratios (a type of number that represents fractions)

- Unlike C++ and Java, Common Lisp has a built-in data type called ***ratio*** that represents fractions ***exactly***, with no rounding error.
  - Ratios are always ***printed in lowest terms*** (but they need not be written in lowest terms): You can write 6/8, but this ratio would be printed as 3/4.
  - If m and n are integers, n is not 0, and n does not divide m, then the value of (/ m n) is a ratio. For example, the value of (/ 12 9) is 12/9 = 4/3.
  - There is no space before or after the / in a ratio: 5/7 ***cannot*** be written as 5 /7 or 5/ 7.
  - In Common Lisp, a number is said to be ***rational*** if it is *either* an integer *or* a ratio.
  - The functions +, -, *, and / accept rational and floating point argument values: If each argument value is rational, the returned result will also be rational; otherwise, the result will be a floating point number.