Functional programming is a style of programming in which the code we write consists of:

- Definitions of *functions that have* ***no side-effects***.
- Expressions that call such programmer-defined functions or call library functions that have *no side-effects*.

We say a function f ***has no side-effects*** if a call of f does nothing except return a value, which implies:

- On return from any call of f, the values stored in variables, data structures, and object components are exactly the same as they were before that call of f.
- This implies execution of f doesn't initialize variables that can be used after f returns (though f may return a new or an existing data object).
- Execution of f does ***not*** do any I/O.
- Execution of f does ***not*** throw an exception.

[It ***isn't*** a side-effect for execution of a function f to change values stored in variables, data structures, or objects that are ***local*** to f, ***but*** the functions used in pure functional programming are ***not*** even allowed to do that!]

Functional programming is a style of programming in which the code we write consists of:

- Definitions of *functions that have **no side-effects***.
- Expressions that call such programmer-defined functions or call library functions that have *no side-effects*.

In (pure) functional programming the bodies of the function definitions we write consist of expressions that ***never*** call functions that have side-effects.

In functional programming, function calls are typically thought of as ***specifying the values they return***.

Examples: One function call might be thought of as specifying ***the sum of the numbers in a list that's passed as its argument***; another function call might be thought of as specifying ***the list of the negative numbers in a list that's passed as its argument***; another function call as specifying ***a list of the 2 roots of the equation*** $ax^2+bx+c = 0$, ***where*** a, b, **and** c ***are the numbers passed as arguments***.

Functional programming is a style of programming in which the code we write consists of:

- Definitions of *functions that have **no side-effects***.
- Expressions that call such programmer-defined functions or call library functions that have *no side-effects*.

In (pure) functional programming the bodies of the function definitions we write consist of expressions that ***never*** call functions that have side-effects.

In functional programming, function calls are typically thought of as ***specifying the values they return.***

We typically ***avoid*** thinking of just how function calls compute the values they return (except when debugging or considering code efficiency)! For example, when reading a recursive function call we typically ***avoid thinking of just how that recursive call computes its value--i.e., we avoid tracing the recursion down***!

Functional programming is a style of programming in which the code we write consists of:

- Definitions of *functions that have* ***<u>no side-effects</u>***.
- Expressions that call such programmer-defined functions or call library functions that have *<u>no side-effects</u>*.

In (pure) functional programming the bodies of the function definitions we write consist of expressions that ***<u>never</u>*** call functions that have side-effects.

In functional programming, function calls are typically thought of as ***specifying the values they return***.

We typically ***avoid*** thinking of just how function calls compute the values they return (except when debugging or considering code efficiency)!

In functional programming, this way of thinking allows us to think of our code as specifying ***<u>what</u>*** is to be computed rather than ***<u>how</u>*** computation will be done.

Advantages of Functional Programming

Since variables, data structures, and objects are immutable, and functions have no side-effects:

1. Code is easier to understand, reason about, and debug.
    - Inaccurate thinking about how and when stored values change can lead to bugs in imperative code.
    - Functional code can also be easier to understand and debug because the flow of information during code execution is simple, limited, and entirely explicit:

      In functional code a function passes information to the functions it calls (via arguments) and returns information (the result) to its caller. That's all!

      Flow of information during execution of imperative code is much more complex: ***Changing a stored value gives updated information to all parts of the code with access to the value***; to understand the code we have to understand just when and how stored values can change and the possible effects of such changes.

Since variables, data structures, and objects are immutable, and functions have no side-effects:

1. Code is easier to understand, reason about, and debug.
2. Code is easier to test.
   - We don't have to worry that a function will update variables and data inappropriately.
   - Functions in functional code are typically ***pure***. A ***pure*** function is one that not only ***has no side-effects*** but also has the property that ***the result it returns depends only on the argument value(s) it receives***: So its result does ***not*** depend on the values of variables and data that exist outside the function.

     A pure function can be tested just by calling it with different argument values and checking the results.

     When programming in a bottom-up style--so each function is written before any function that calls it--using only pure functions, each function can be tested in this way ***as soon as it has been written***, which is a good practice because:

Since variables, data structures, and objects are immutable, and functions have no side-effects:

1. Code is easier to understand, reason about, and debug.
2. Code is easier to test.
   - We don't have to worry that a function will update variables and data inappropriately.
   - Functions in functional code are typically ***pure***.

A pure function can be tested just by calling it with different argument values and checking the results.

When programming in a bottom-up style--so each function is written before any function that calls it--using only pure functions, each function can be tested in this way ***as soon as it has been written***, which is a good practice because:

Since variables, data structures, and objects are immutable, and functions have no side-effects:

1. Code is easier to understand, reason about, and debug.
2. Code is easier to test.
    - We don't have to worry that a function will update variables and data inappropriately.
    - Functions in functional code are typically ***pure***.

      A pure function can be tested just by calling it with different argument values and checking the results.

      When programming in a bottom-up style--so each function is written before any function that calls it--using only pure functions, each function can be tested in this way ***as soon as it has been written***, which is a good practice because:

      (a) Bugs are easier to fix when they're detected before you've forgotten details of how the code is supposed to work!
      (b) It reduces the risk of forgetting to test some functions.
      (c) Detected bugs will likely be in the function being tested, as each function it calls has *already* been tested.

Since variables, data structures, and objects are immutable, and functions have no side-effects:

1. Code is easier to understand, reason about, and debug.
2. Code is easier to test.
   - We don't have to worry that a function will update variables and data inappropriately.
   - Functions in functional code are typically ***pure***.

     A pure function can be tested just by calling it with different argument values and checking the results.

     When programming in a bottom-up style--so each function is written before any function that calls it--using only pure functions, each function can be tested in this way ***as soon as it has been written***.
3. Code can be automatically parallelized.
   - Different arguments of a function call can be evaluated in parallel, as evaluation of one argument expression cannot interfere with or affect evaluation of another.

Since variables, data structures, and objects are immutable, and functions have no side-effects:

1. Code is easier to understand, reason about, and debug.
2. Code is easier to test.
3. Code can be automatically parallelized.

However, when considering benefits of functional programming we should also bear in mind that immutability does have a cost:

When executed on computer systems that are in common use today, programs written in a functional style are generally less efficient (are slower and use more memory) than equivalent imperative programs.

- For example, imperative code that frequently updates individual elements of a large array cannot, in general, be replaced with similarly efficient functional code.

Using Lisp for Functional Programming

Getting Started with the CLISP Common Lisp Interpreter

- While logged in to **mars**, 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.

- The I/O done by the ***read*** and ***print*** steps is done ***by the Lisp interpreter*** (which ***isn't*** written in a purely functional style) and ***not*** by the expression you enter. If the expression is written in a purely functional style, its evaluation does no I/O!

Getting Started with the CLISP Common Lisp Interpreter

- While logged in to **mars**, 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 - 4.1f	Lisp: (- 3 4.1)
Java: 3 - 4.1f + 2	Lisp: (+ (- 3 4.1) 2)
Java: Math.sqrt(3*4.1f)	Lisp: (sqrt (* 3 4.1))

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:

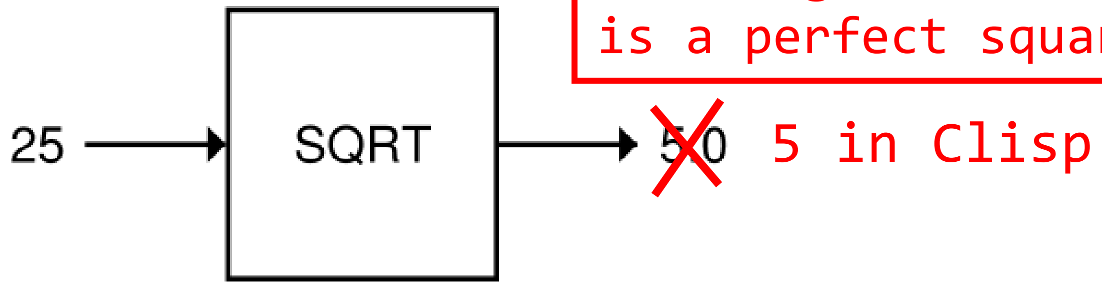
Java: 3 – 4.1f	Lisp: (– 3 4.1)
Java: 3 – 4.1f + 2	Lisp: (+ (– 3 4.1) 2)
Java: Math.sqrt(3*4.1f)	Lisp: (sqrt (* 3 4.1))

  - 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.

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.



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

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 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.