Reasons to Use Lisp
Rather Than Java or C++ For
Functional Programming in This Course

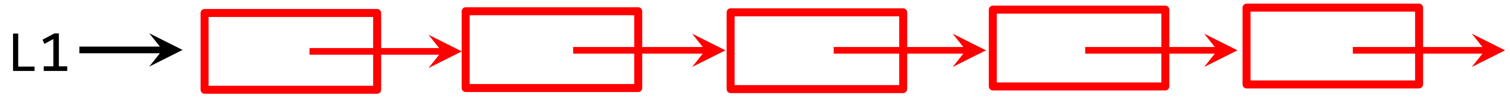
Java and C++ were designed for imperative programming, and are not ideally suited to functional programming.

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*:**



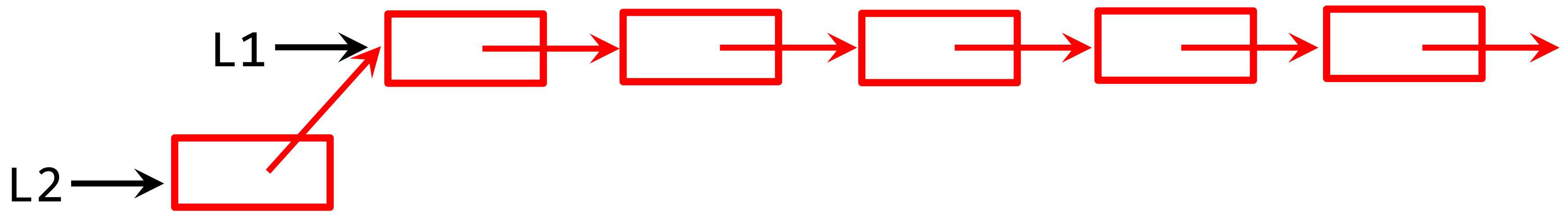
Java and C++ were designed for imperative programming, and are not ideally suited to functional programming.

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*:**



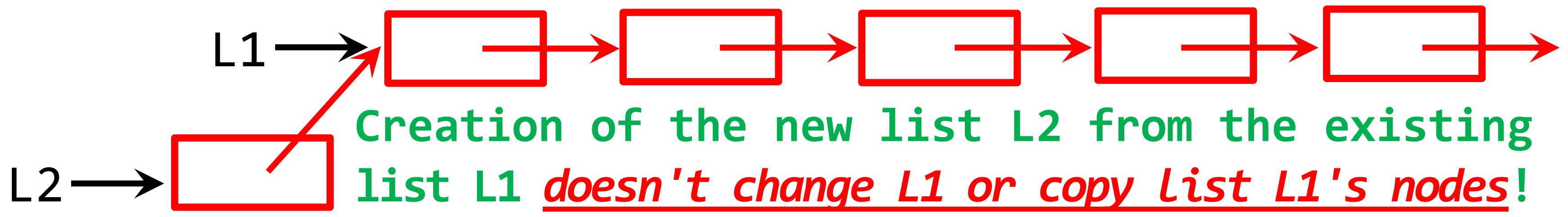
Java and C++ were designed for imperative programming, and are not ideally suited to functional programming.

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*:**



Creation of the new list L2 from the existing list L1 *doesn't change L1 or copy list L1's nodes*!

NOTE: C++ singly-linked lists (`std::forward_list`) don't support this operation!

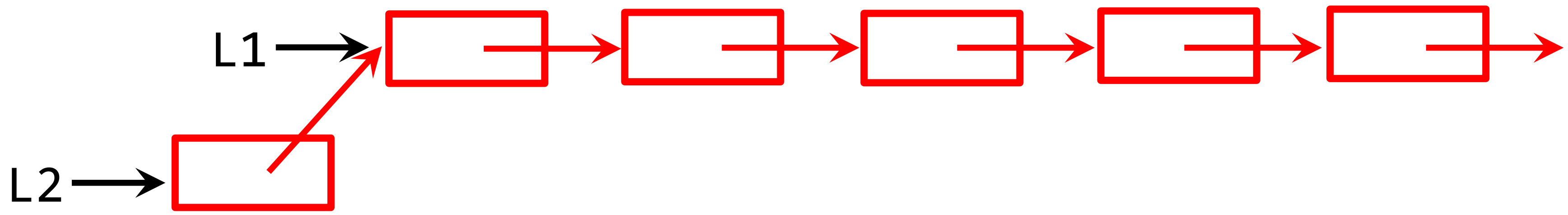
Java and C++ were designed for imperative programming, and are not ideally suited to functional programming.

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*:**



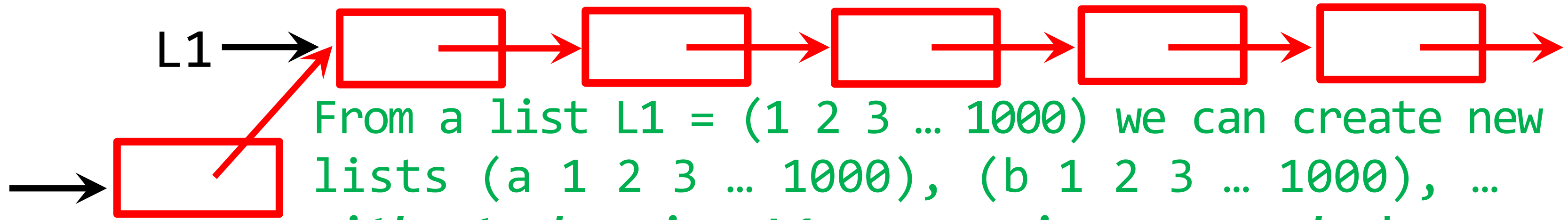
Java and C++ were designed for imperative programming, and are not ideally suited to functional programming.

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*:**



From a list L1 = (1 2 3 … 1000) we can create new lists (a 1 2 3 … 1000), (b 1 2 3 … 1000), … ***without changing L1 or copying any nodes***!

NOTE: C++ singly-linked lists (`std::forward_list`) don't support this!

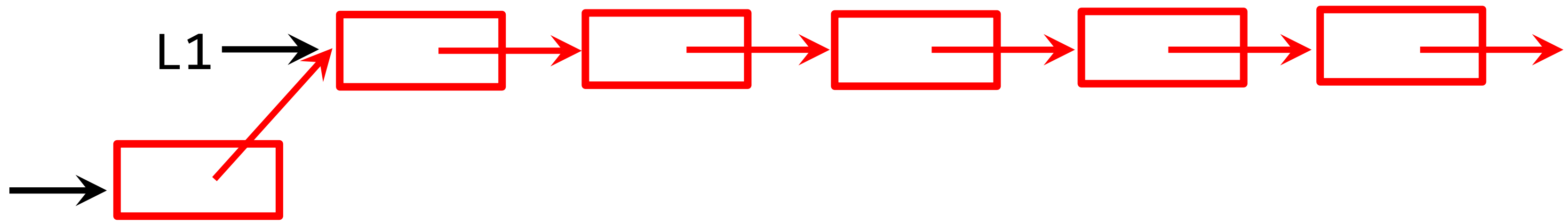
Java and C++ were designed for imperative programming, and are not ideally suited to functional programming.

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*:**



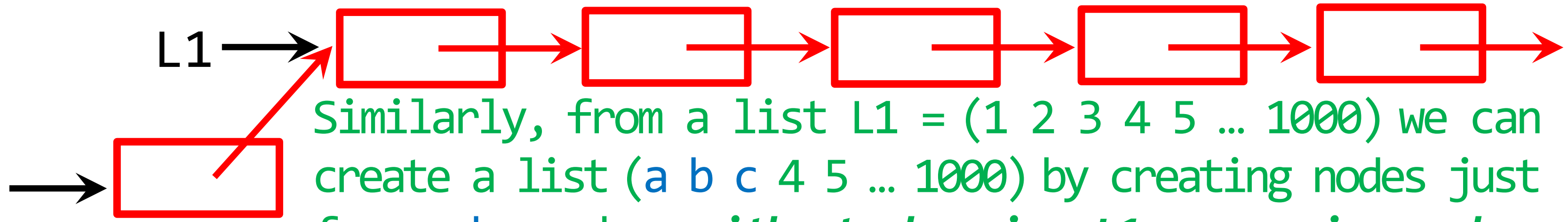
Java and C++ were designed for imperative programming, and are not ideally suited to functional programming.

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*:**



Similarly, from a list L1 = (1 2 3 4 5 … 1000) we can create a list (a b c 4 5 … 1000) by creating nodes just for a, b, and c, ***<u>without changing L1 or copying nodes</u>***.

NOTE: C++ singly-linked lists (`std::forward_list`) don't support this!

Java and C++ were designed for imperative programming, and are 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 data type, and some predefined functions that perform common operations on such lists to create new singly-linked lists ***<u>without changing existing lists</u>***.

Java and C++ were designed for imperative programming, and are 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:

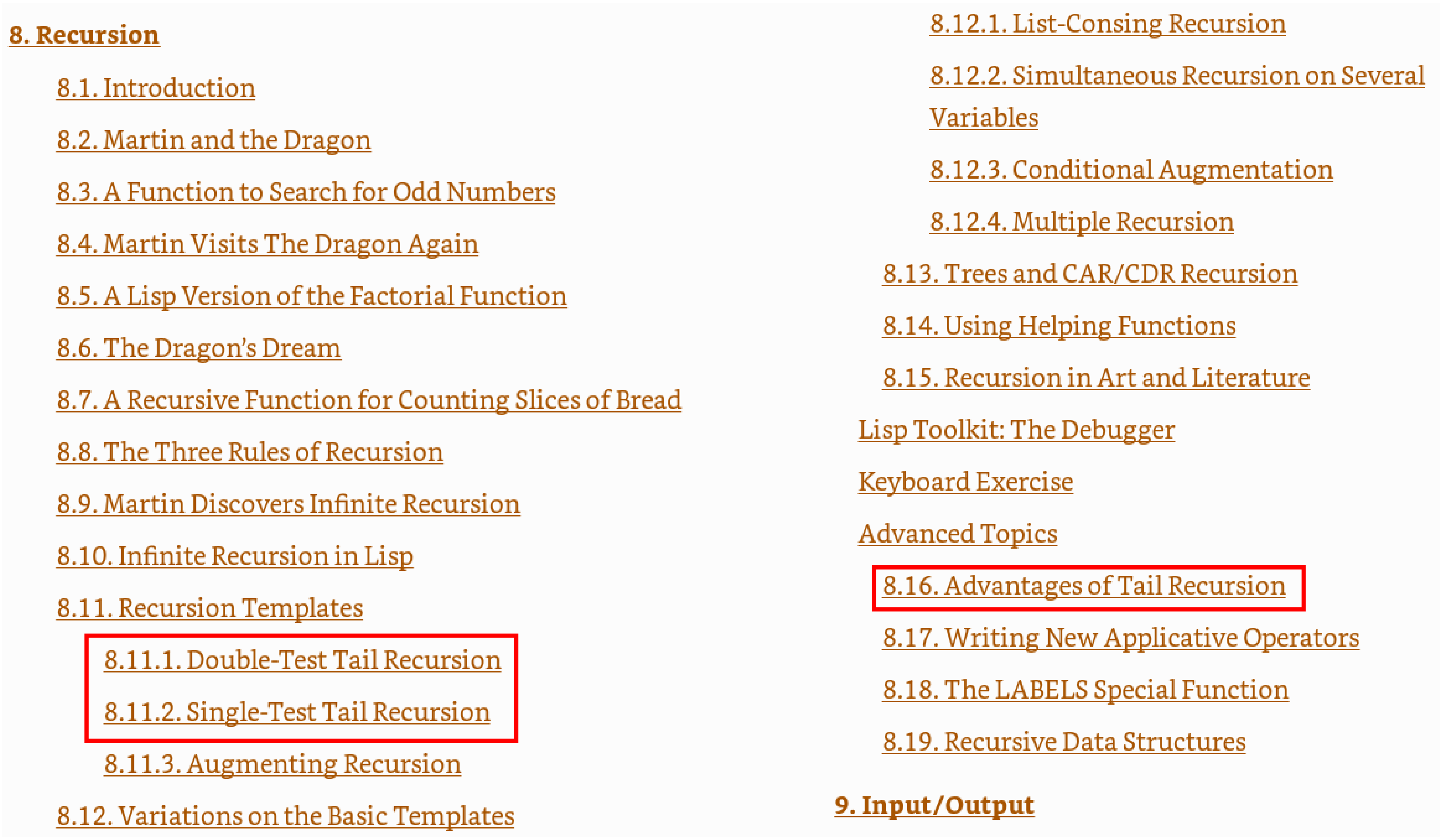
- Java has no predefined singly-linked list class.

  Many languages designed to support functional programming have a predefined singly-linked list data type, and some predefined functions that perform common operations on such lists to create new singly-linked 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.

From the Table of Contents of Touretzky's book:



Java and C++ were designed for imperative programming, and are 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 data type, and some predefined functions that perform common operations on such lists to create new singly-linked lists ***<u>without changing existing lists</u>***.

- Java does ***<u>not</u>*** 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.

Java and C++ were designed for imperative programming, and are 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 data type, and some predefined functions that perform common operations on such lists to create new singly-linked 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 or C++ in a purely functional style.

But it’s common in Java and C++ to use *higher-order functions** (especially functions that take functions as arguments) and to use *lambda expressions*--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 or C++ in a purely functional style.

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

It's currently uncommon to use Java or C++ in a purely functional style.

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

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

- Lisp has a built-in singly-linked list data type and many predefined functions for working with such lists that create new lists ***without changing existing lists***.
- Lisp declarations of higher-order functions are usually considerably less verbose than corresponding Java or C++ declarations, and Lisp code that uses such functions tends to be less brittle than corresponding Java or C++ code.
  Moreover, compiler error messages relating to higher-order functions in Java or C++ code are quite often verbose and can sometimes be hard to decipher.
- Most Lisp implementations that are in common use have support for tail recursion optimization.

Background Information About Lisp

- The first version of Lisp was designed by McCarthy at MIT in 1958; among high-level languages that have seen widespread use, Lisp is the 2nd oldest (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 about 30 years (early 60s – early 90s) Lisp was the dominant programming language in the AI research community.
- "Lisp" is an acronym for LISt Processor: Lisp's principal built-in data structure is the list (e_1 … e_n). Here ***any of the*** e***'s may also be a list***, so arbitrarily complex data structures can be implemented as lists.
- It's common for most of the data structures in a Lisp program to be lists, especially when Lisp is used in a functional style and in early versions of Lisp programs. (In this course, the only Lisp data types you will use in your functions are numbers, symbols, and lists.)

- “Lisp” is an acronym for LISt Processor: Lisp's principal built-in data structure is the list (e_1 … e_n). Here ***any of the* e*'s may also be a list*,** so arbitrarily complex data structures can be implemented as lists.
- Lisp *code* also takes the form of lists! A consequence of this is that Lisp has many parentheses and looks quite different from other languages. A more important consequence is: ***Lisp can easily analyze and manipulate Lisp-like code as Lisp data.*** (The "code and data have the same form" property of Lisp is called ***homoiconicity***.)
- Lisp macros--which are considered by many advanced Lisp programmers to be one of the very best features of the language--take advantage of the fact that Lisp can easily analyze and manipulate Lisp code.

  Macros allow Lisp programmers to add new language constructs to Lisp: Lisp is ***programmer-extensible***. (Footnote 5 in ch. 7 of Seibel says more about this.)