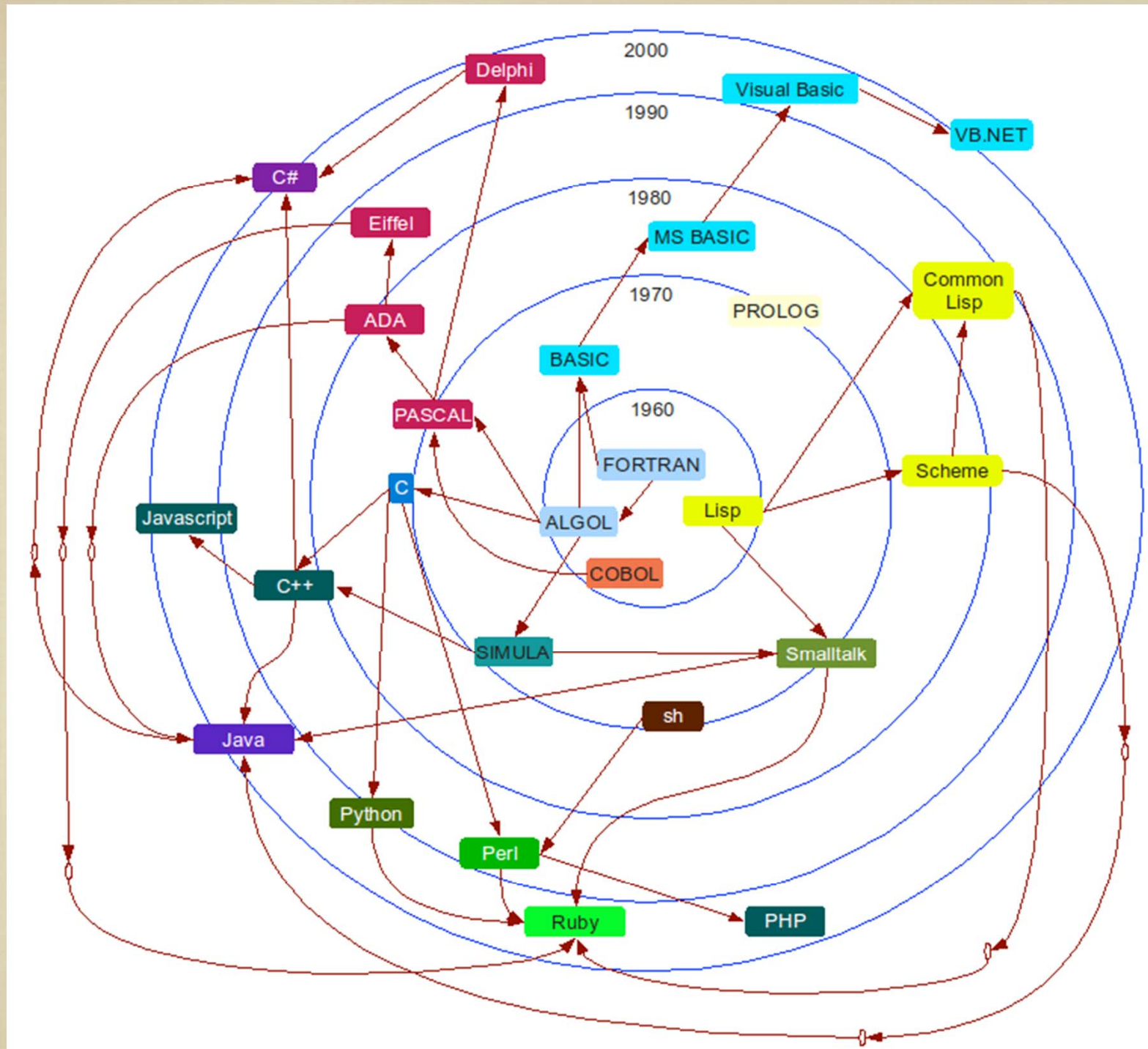
Functional Programming I

Spring 2014

Carola Wenk


2000
1990
1980
1970
1960
Delphi
Visual Basic
VB.NET
C#
Eiffel
MS BASIC
Common Lisp
PROLOG
ADA
BASIC
PASCAL
FORTRAN
Scheme
C
Javascript
ALGOL
Lisp
COBOL
C++
SIMULA
Smalltalk
sh
Java
Python
Perl
Ruby
PHP

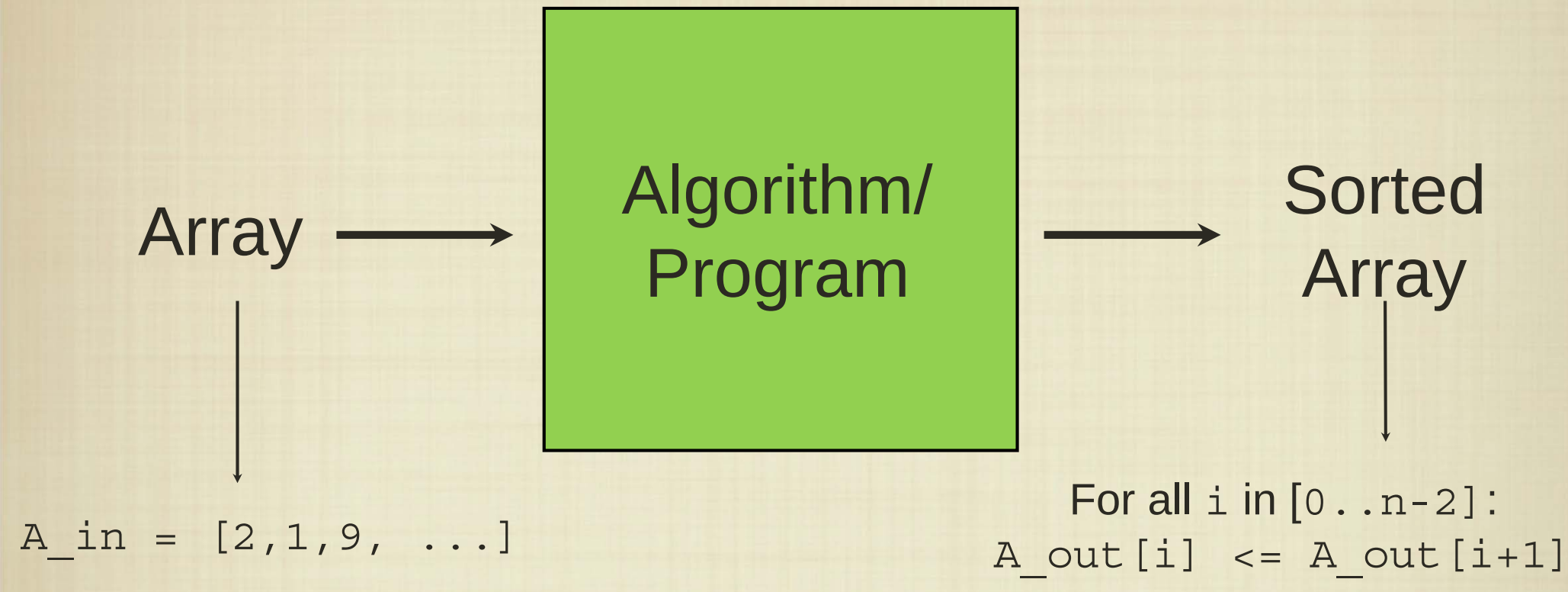
What we've seen so far

```
def f(a, b):
  print "this is function f"
  return a+b;

x = 1; y = 2; evens = 0; odds = []
print f(1, 2)
print f('z', f('a', 'b'))
for i in range(1,10):
      if (i % 2 == 0):
         evens += i
      else:
         odds.append(i)
print evens; print sum(odds)
```

```
#include <stdio.h>

struct my_node {
   int data;
   my_node* next;
};

int main() {
   my_node* p, q;

   p = new my_node;
   p -> data = 15;
   q = p;
   free(p);
   q -> data = 99;
   return 0;
}
```

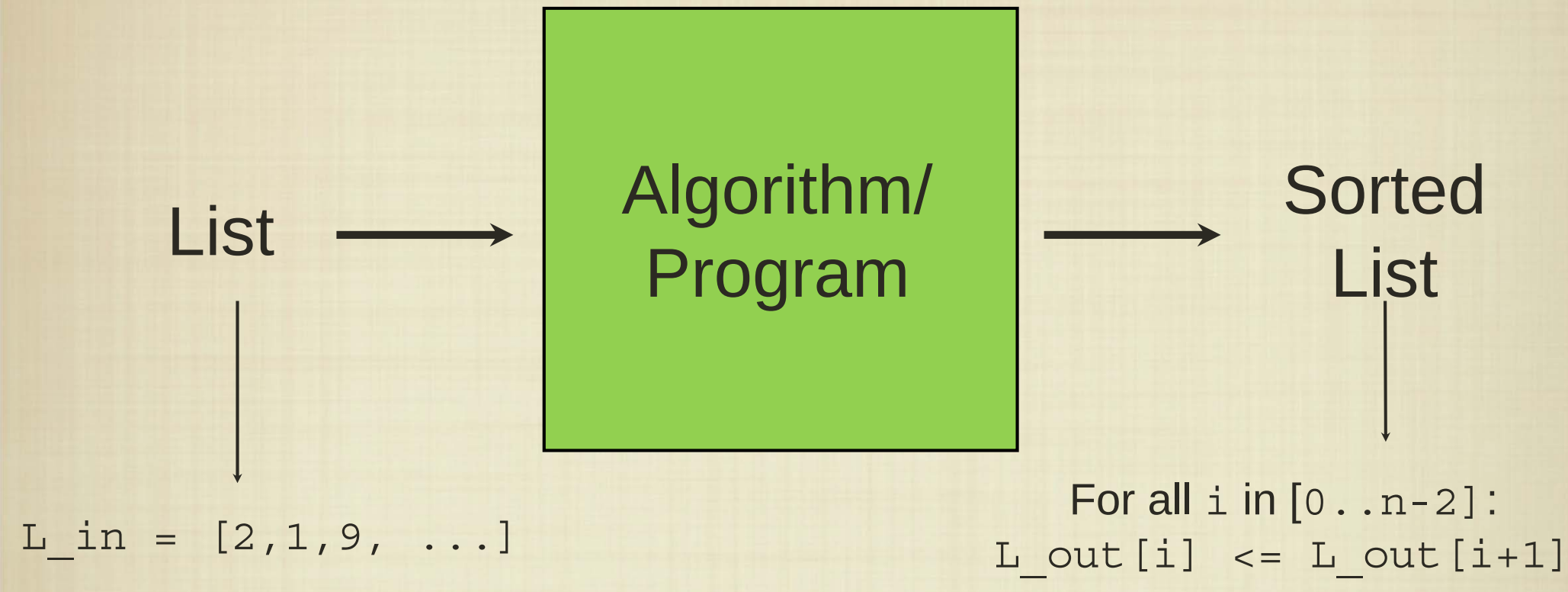
- In Python, Java, C/C++, a program went about its business by executing a sequence of statements (in a function, loop, if statement, etc).

Sorting Specifications



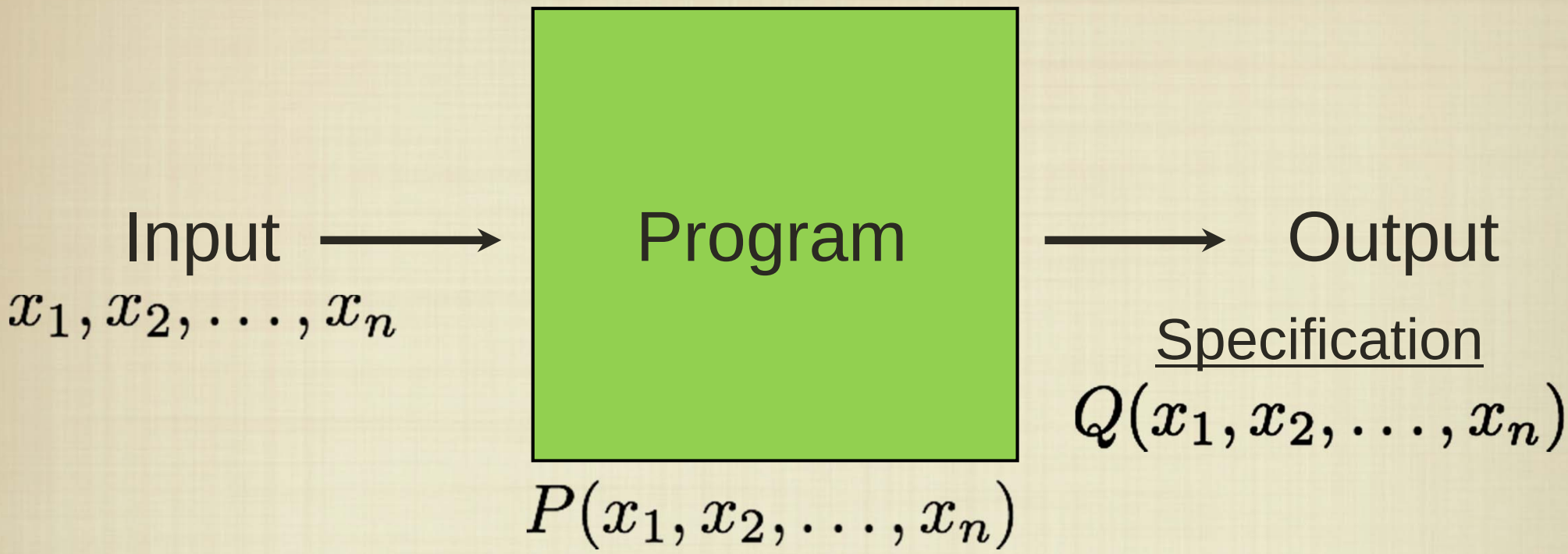
We want to specify an input, and what the output should look like, using first-order logic.

Sorting Specifications



We want to specify an input, and what the output should look like, using first-order logic.

Program Execution and Logic



For our purposes, we can view program execution as the application of a (complicated) logical formula to the given input.

When the output specification is guaranteed to follow from any execution (i.e., for all executions), we say the program is correct.

Program Execution and Logic

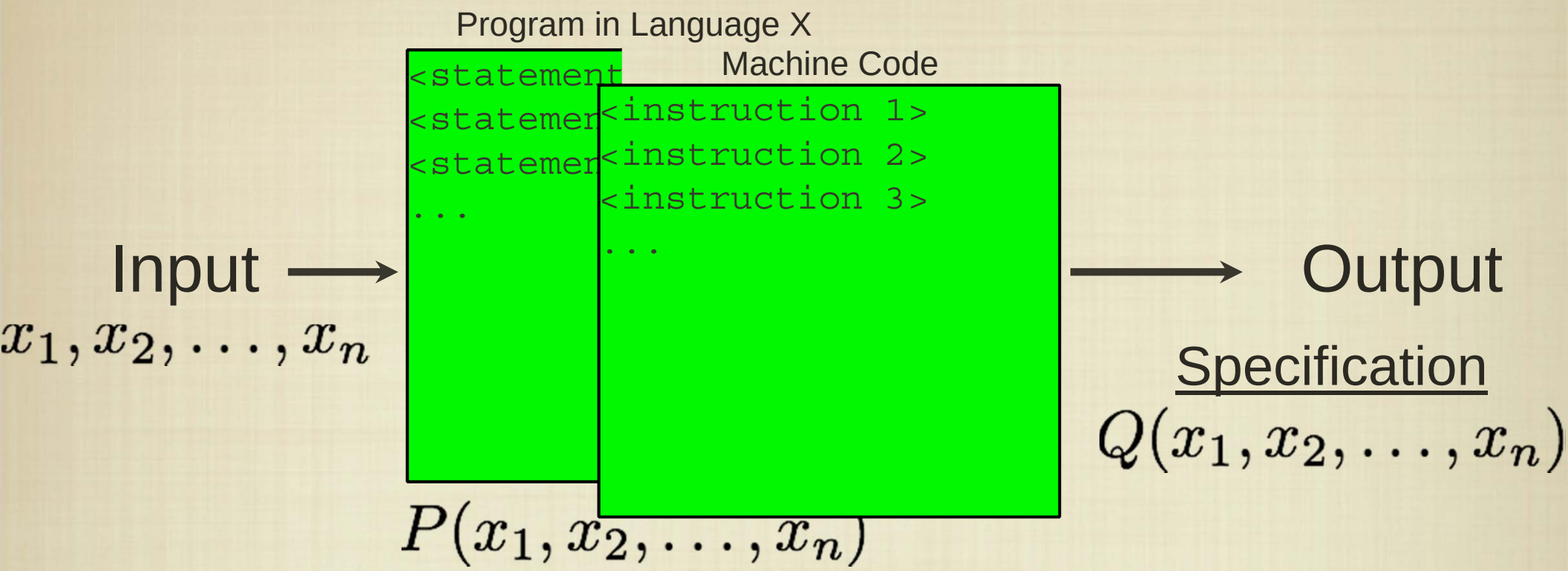
$$P(x_1, x_2, \ldots, x_n) \stackrel{?}{\rightarrow} Q(x_1, x_2, \ldots, x_n)$$

So, there is a natural connection between a logical specification for the output and the program itself (regardless of the language).

Deriving the formula for a computer program is somewhat cumbersome -- we will use other techniques to prove this implication.

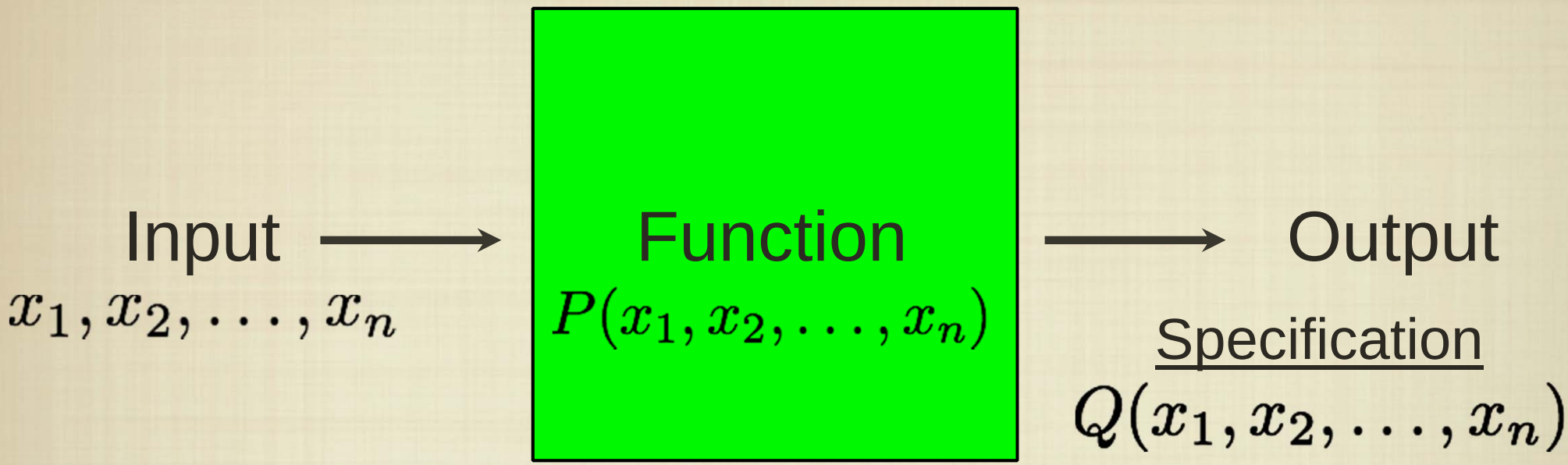
What does testing a program on selected inputs prove?

Program Execution and Logic



- For a particular language, we focused on arguing the input was transformed in such a way that it satisfied some logical property at the end of execution. Then, we argued that this property implied correctness.
- Our (correct) program is translated to machine code that follows roughly the same pattern (if-then, conditional, assignment, etc.).
- But something doesn't match - we're trying to associate a logical function with the program. Is Python/Java/C/C++ the best way?

In An Ideal World



Ideally we'd be able to just give a function instead of a program, and not worry about writing a sequence of statements that produce the right $P(x_1, x_2, \ldots, x_n)$.

Functional Programming

- Define a function, and let the runtime system do the work.
- A functional programming language must be extremely high level, and so it is usually even more highly-managed than Python.
- What would the syntax of such a language look like? Would we use pure logic?

Scheme

- Scheme is based on the LISP language, developed in 1958. It is actually the second oldest programming language!

- Scheme uses “Polish” (i.e., prefix) notation:

```
(* 3 (+ (+ 1 (* 2 4)) (- 7 1)))
```

- In (pure) Scheme, “everything is a list”, and there is no concept of “sequential” execution. Also, instead of variable assignment, generally, variable binding is used.

```
(define (f n)
  (if (= n 0)
      1
      (* n (f (- n 1)))))
```

Running Scheme Programs

Scheme is interpreted, and so there is no “main” method, we simply call functions as needed.

Scheme source code is simply a collection of (possibly interdependent) functions, followed by function evaluations

Each function, in reality, is just a nested (linked) list.

```
(define (f n) (if (= n 0) 1 (* n (f (- n 1)))))
```

Running Scheme Programs

Scheme is interpreted, and so there is no “main” method, we simply call functions as needed.

Scheme source code is simply a collection of (possibly interdependent) functions, followed by function evaluations

Each function, in reality, is just a nested (linked) list.

```
(define (f n) (if (= n 0) 1 (* n (f (- n 1)))))
```

Running Scheme Programs

Scheme is interpreted, and so there is no “main” method, we simply call functions as needed.

Scheme source code is simply a collection of (possibly interdependent) functions, followed by function evaluations

Each function, in reality, is just a nested (linked) list.

```
(define (f n) (if (= n 0) 1 (* n (f (- n 1)))))
```

Running Scheme Programs

Scheme is interpreted, and so there is no “main” method, we simply call functions as needed.

Scheme source code is simply a collection of (possibly interdependent) functions, followed by function evaluations

Each function, in reality, is just a nested (linked) list.

```
(define (f n) (if (= n 0) 1 (* n (f (- n 1)))))
```

Running Scheme Programs

Scheme is interpreted, and so there is no “main” method, we simply call functions as needed.

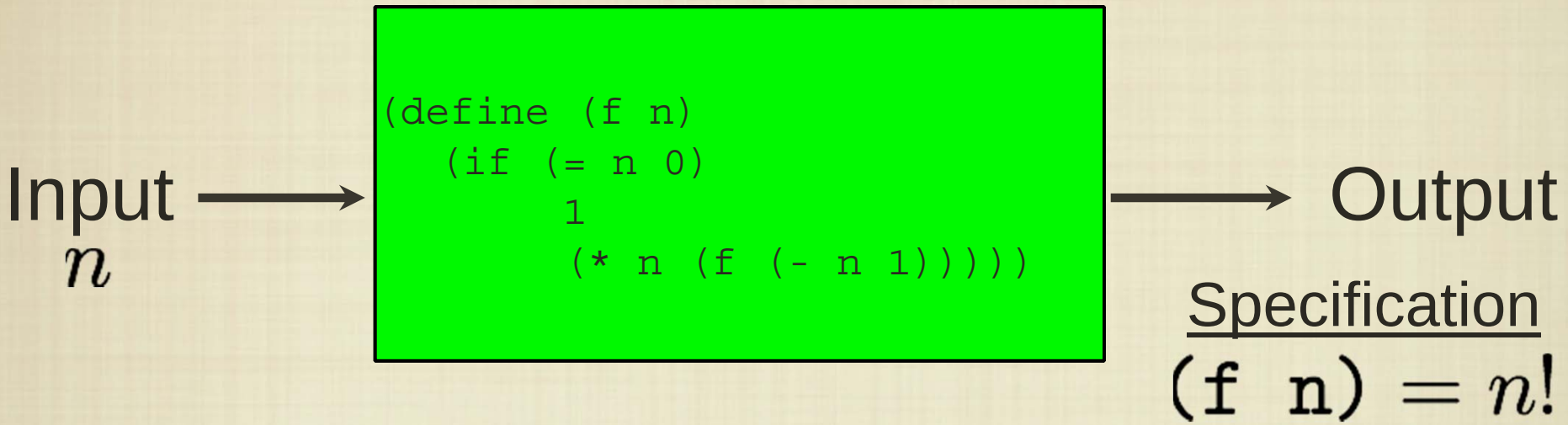
Scheme source code is simply a collection of (possibly interdependent) functions, followed by function evaluations

Each function, in reality, is just a nested (linked) list.

```
(define (f n) (if (= n 0) 1 (* n (f (- n 1)))))
```

Variable scope is defined by nesting, and a function is really just a list.

How Do We Prove Correctness?



Base Case: `(f 0)` = 1 = 0!

Inductive Step: Suppose that `(f (- n 1))` = $(n-1)!$ Then, since we're multiplying by `n`, `(f n)` = $n!$

List Manipulation

- Of course, more sophisticated algorithms will require us to access parts of a list.
- The `cons` function prepends an element to a list.
- The `car` function returns the first element of a list.
- The `cdr` function removes the first element of a list, and returns the remaining list.
- These basic functions are used to implement all of the list operations we've seen (e.g. indexing and slicing), and many of these are implemented in the Scheme standard library.