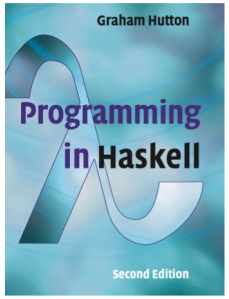
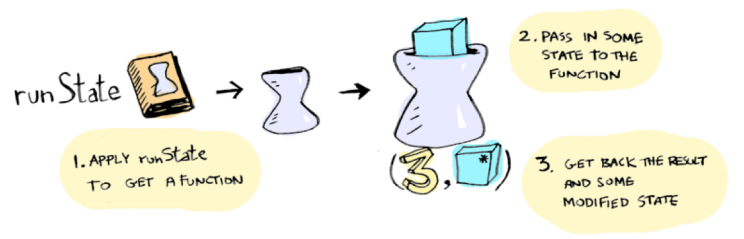
CS 320: Concepts of Programming Languages

Wayne Snyder
Computer Science Department
Boston University

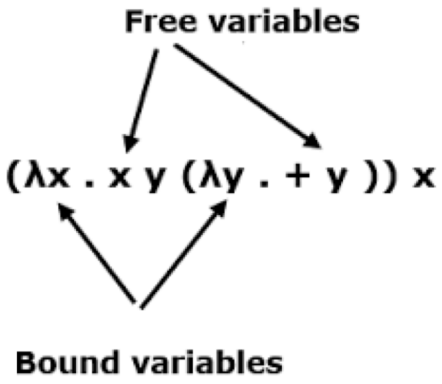


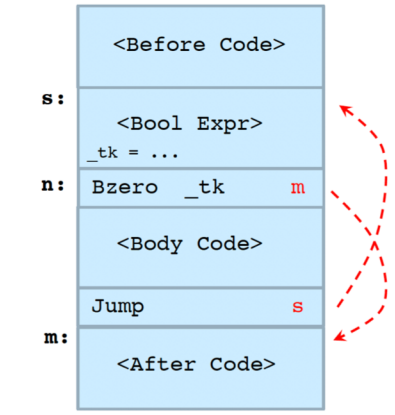



```
second :: [a] -> a
second xs = head (tail xs)

swap :: (b, a) -> (a, b)
swap (x,y) = (y,x)

pair :: a -> b -> (a, b)
pair x y = (x,y)
```





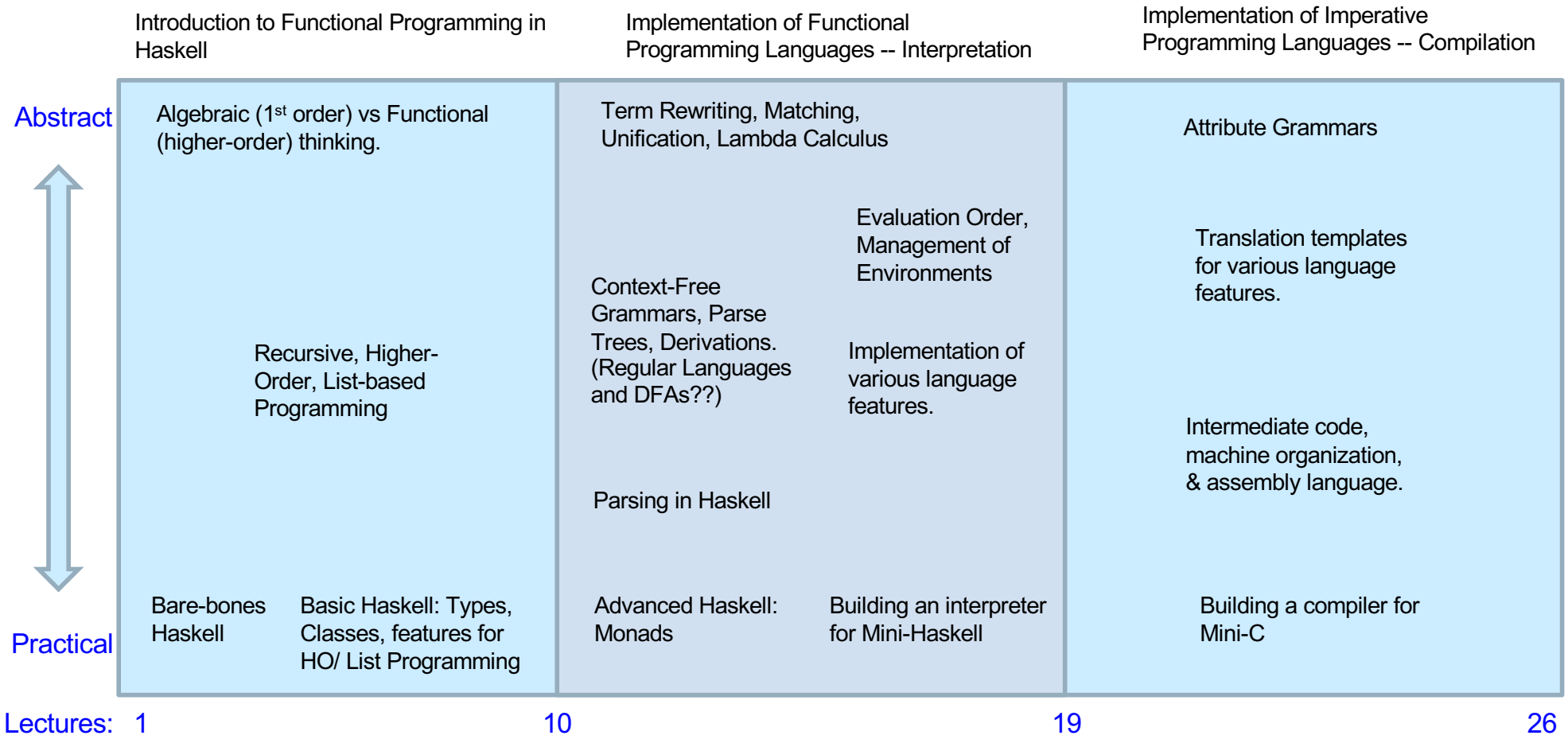
(While expr stmt)

Schedule of Topics

26 lectures: 2 midterms, 1 intro lecture, so 23 lectures

Approximately 8 lectures = 1 month for each major area:

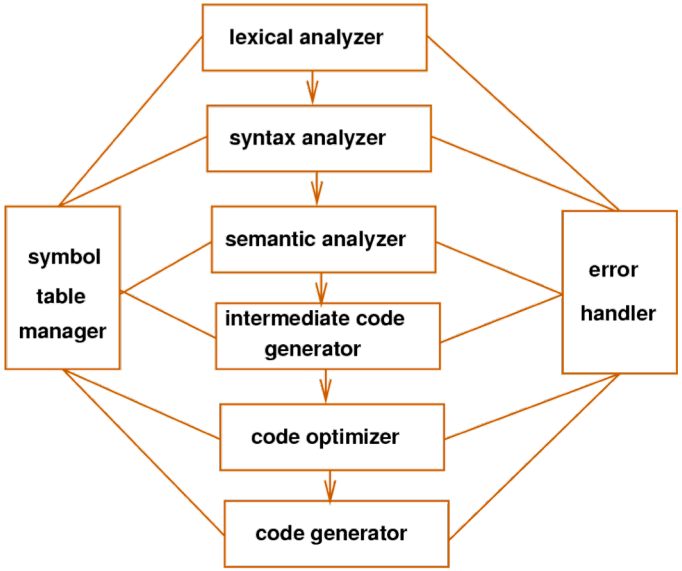
- 8 lectures (2 – 9) for Introduction to Functional Programming in Haskell, followed by midterm 1 (lecture slot 10)
- 8 lectures (11 – 18) for Interpreters: Implementation of Functional Programming Languages, plus further development of Haskell, particularly monads and parsing, followed by midterm 2 (lecture slot 19)
- 7 lectures (20 – 26) for Compilers: Implementation of Imperative Programming Languages.



What is a Programming Language?

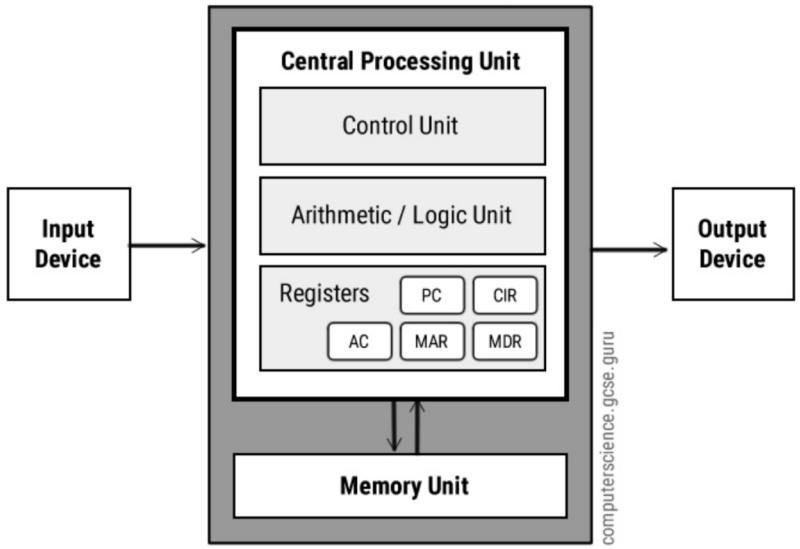
A Programming Language is defined by

- **Syntax** – The rules for putting together symbols (e.g., Unicode) to form expressions, statements, procedures, functions, programs, etc. It describes legal ways of specifying computations.
- **Semantics** – What the syntax *means* in a particular model of computation. This can be defined abstractly, but is implemented by an algorithm + data structures (examples: Python interpreter or C compiler).



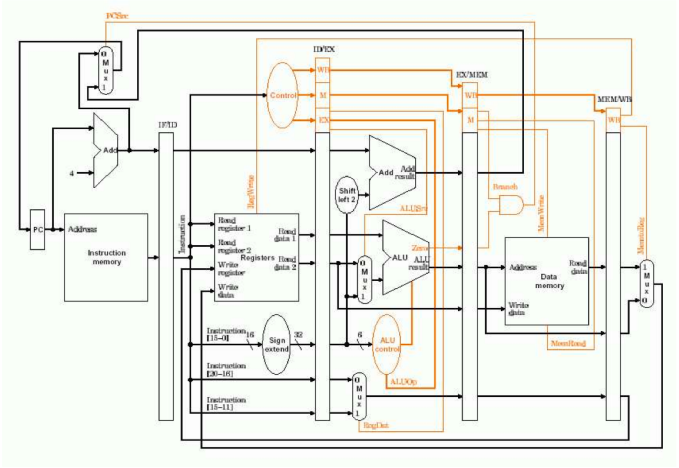
What is a Programming Language?

Modern digital computers are based on the Von Neumman Architecture of 1945:



Execution Cycle:

1. Read next instruction and data from memory;
2. Process the data according to the instruction in the ALU;
3. Write the result to memory;
4. Go to 1

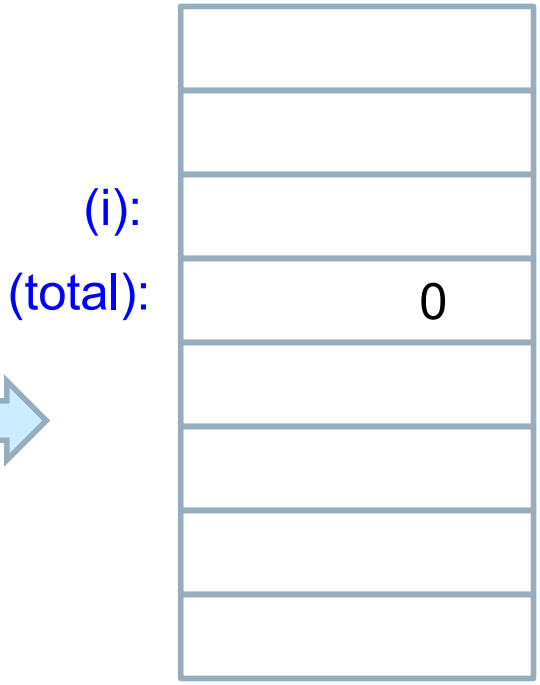


Imperative Programming Languages

An imperative language sticks close to this model, and the semantics is all about what happens in the memory. It therefore emphasizes statements which have an effect on memory.

The semantics of the following loop in C is the incremental changes to the variables `i` and `total` that represent locations in memory:

```
/* Add nums 1 to 10 */
total = 0;
for (i=1; i≤10; ++i)
   total=total+i;
```

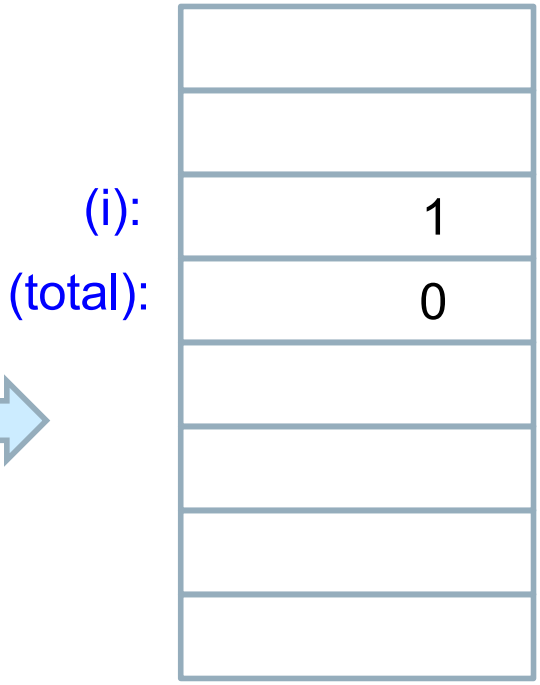


Imperative Programming Languages

An imperative language sticks close to this model, and the semantics is all about what happens in the memory. It therefore emphasizes statements which have an effect on memory.

The semantics of the following loop in C is the incremental changes to the variables `i` and `total` that represent locations in memory:

```
/* Add nums 1 to 10 */
total = 0;
for (i=1; i≤10; ++i)
   total=total+i;
```

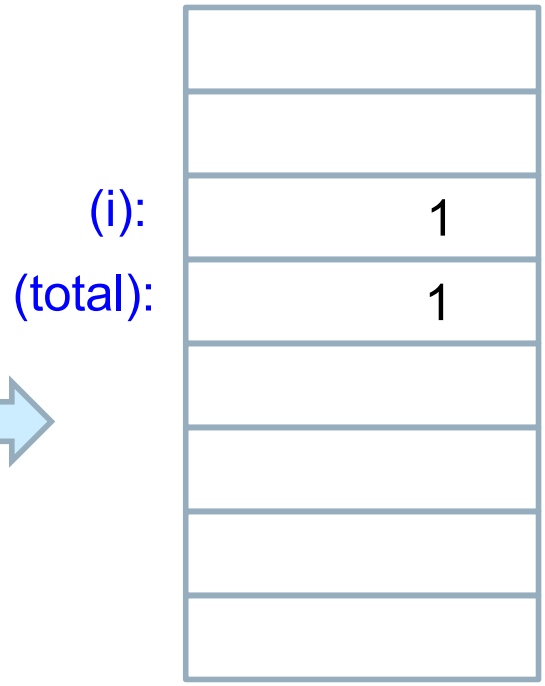


Imperative Programming Languages

An imperative language sticks close to this model, and the semantics is all about what happens in the memory. It therefore emphasizes statements which have an effect on memory.

The semantics of the following loop in C is the incremental changes to the variables `i` and `total` that represent locations in memory:

```
/* Add nums 1 to 10 */
total = 0;
for (i=1; i≤10; ++i)
   total=total+i;
```

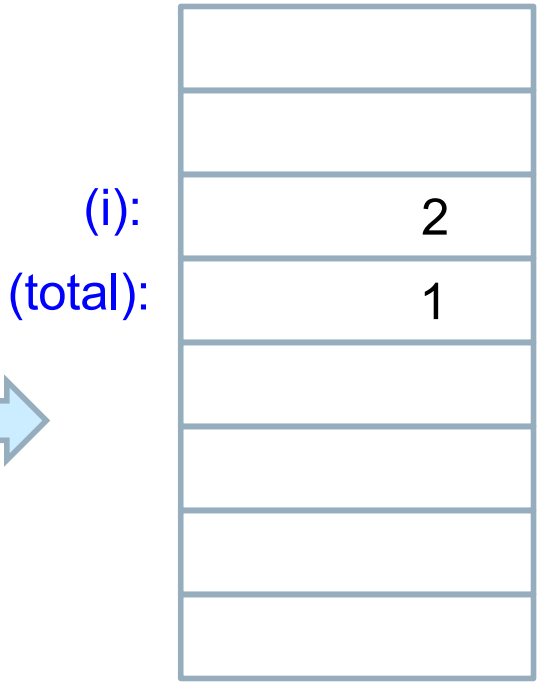


Imperative Programming Languages

An imperative language sticks close to this model, and the semantics is all about what happens in the memory. It therefore emphasizes statements which have an effect on memory.

The semantics of the following loop in C is the incremental changes to the variables `i` and `total` that represent locations in memory:

```
/* Add nums 1 to 10 */
total = 0;
for (i=1; i≤10; ++i)
   total=total+i;
```

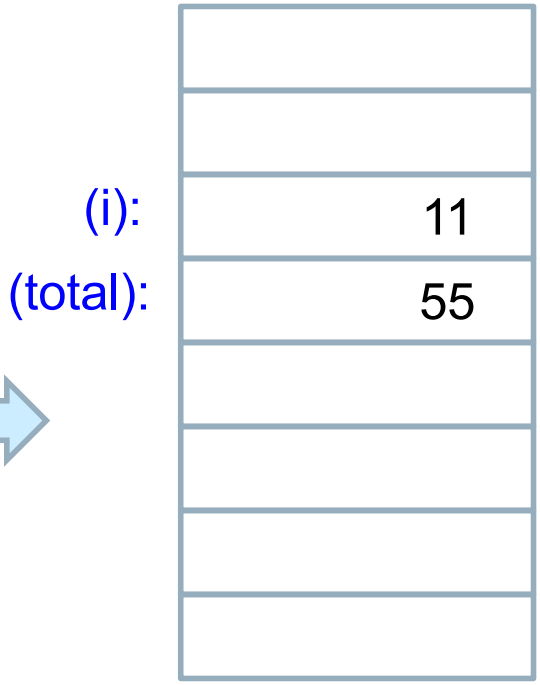


Imperative Programming Languages

An imperative language sticks close to this model, and the semantics is all about what happens in the memory. It therefore emphasizes statements which have an effect on memory.

The semantics of the following loop in C is the incremental changes to the variables `i` and `total` that represent locations in memory:

```
/* Add nums 1 to 10 */
total = 0;
for (i=1; i≤10; ++i)
   total=total+i;
```



Digression on Terminology:

In programming language theory (and in this course) the memory, as a whole, and in whatever form and technology it involves is called the state, so just remember:

State $=_{def}$ All the data stored in memory

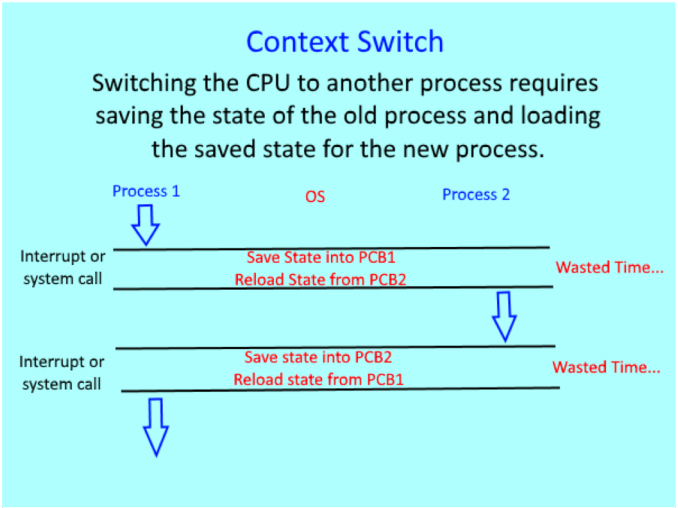
as in "state of the computation" and "statement." Therefore, the meaning of an imperative language is the sequence of state transitions that the statements in the language produce.

Such a process is therefore called stateful.

The paradigmatic programming language feature which affects state is the assignment statement:

```
x = 4 * y + 1;
```

Compare: Context Switch



(End of Digression!)

What's Wrong with the Imperative (Stateful) Paradigm?

- Memory is complicated (registers, status bits, caches, RAM, run-time stack, …) and does not always correspond in a natural way to program structure.
- Mentally keeping track of dozens or 100's of variables is difficult.
- Debuggers (Eclipse, GDB) help, but are very complicated (many people just use print statements to print out all relevant variables)
- It feels more like accounting than mathematics!

```
 1 public static boolean binarySearch(int[] A, int k, int left, int right) {
 2
 3     db("Looking for " + k + " in A[" + left + ".." + right + "]");
 4
 5     if(left > right)
 6       return false;
 7
 8     int middle = (left + right) / 2;
 9     db("Comparing with " + A[middle]);
10     if(A[middle] == k)
11         return true;
12     else if(k < A[middle])
13         return binarySearch(A, k, left, middle - 1);
14     else
15         return binarySearch(A, k, middle + 1, right);
16 }
```

Again, if we print out the array and number first, and then trace the program with `debug = true`, the output will be:

```
number = 15
A = [2, 7, 12, 15, 19, 25, 26, 38, 45, 78]
        Looking for 15 in A[0..9]
        Comparing with 19
        Looking for 15 in A[0..3]
        Comparing with 7
        Looking for 15 in A[2..3]
        Comparing with 12
        Looking for 15 in A[3..3]
        Comparing with 15
Number 15 found!
```

How to eliminate or at least control the notion of state?

Let us consider: what is the essence of mathematical reasoning, for example in the case of ordinary algebra?

Example:

```
    (3 + 4) * (5 - 2)
=>  7 * (5 - 2)
=>  7 * 3
=>  21
```

Note:

There are no variables keeping track of the state.

The computation rewrites one term to another using the basic definitions of + and *.

Example 2: Prove the following equivalence in Boolean Logic:

$$(A \wedge C) \vee (\neg A \wedge \neg B \wedge C) \equiv (A \wedge C) \vee (\neg B \wedge C)$$

Proof (using the distributive law):

$$\begin{aligned}
&(A \wedge C) \vee (\neg A \wedge \neg B \wedge C) \\
&\equiv (A \vee (\neg A \wedge \neg B)) \wedge C \\
&\equiv ((A \vee \neg A) \wedge (A \vee \neg B)) \wedge C \\
&\equiv (True \wedge (A \vee \neg B)) \wedge C \\
&\equiv (A \vee \neg B) \wedge C \\
&\equiv (A \wedge C) \vee (\neg B \wedge C)
\end{aligned}$$

Note:

There are no variables keeping track of the state.

The proof rewrites one term to another using axioms (e.g., the distributive law).

Functional Programming Languages

Functional programming is a style of programming in which the fundamental method of computation is applying a function to arguments; a functional programming language is one that supports and encourages the functional style.

It almost always leads to simpler programs in which you can think like a mathematician, not an accountant!

C (imperative):

```
/* Add nums 1 to 10 */
total = 0;
for (i=1; i≤10; ++i)
   total=total+i;
```

Haskell (functional):

```
sum [1 .. 10]
```

Functional Programming Languages

In essence, functional programming is stateless, because it avoids storing data in variables, and the role of memory is hidden from the programmer.

The general term for this property is Referential Transparency:

"**Referential transparency** and **referential opacity** are properties of parts of computer programs. An expression is called referentially transparent if it can be replaced with its corresponding value without changing the program's behavior. This requires that the expression is pure, that is to say the expression value must be the same for the same inputs and its evaluation must have no side effects. An expression that is not referentially transparent is called referentially opaque." - Wikipedia

Functional Programming Languages

"In mathematics all function applications are referentially transparent, by the definition of what constitutes a mathematical function. The importance of referential transparency is that it allows the programmer and the compiler to reason about program behavior as a rewrite system. This can help in proving correctness, simplifying an algorithm, assisting in modifying code without breaking it, or optimizing code by means of memoization, common subexpression elimination, lazy evaluation, or parallelization."

```
int globalValue = 0;

int rq(int x) {
  globalValue++;
  return x + globalValue;
}

int rt(int x) {
  return x + 1;
}
```

Rq is NOT referentially transparent, due to the global variable:

What is value of

rq(5) ? rq(1) * rq(2) ?

rt(5) ? rt(1) * rt(2) ?

Functional Programming Languages

Functional programming languages have been developed in parallel with imperative languages since the 1950's.

Functional programming has been supported by many languages (e.g., Python, Javascript).