Spring 2019

CSE 216 : Programming Abstractions

TOPIC 1 – PROGRAMMING PARADIGMS

Machine Instructions

A machine instruction consists of several bytes in memory that tell the processor to perform one machine operation.

The processor looks at machine instructions in main memory one after another, and performs one machine operation for each machine instruction.

The collection of machine instructions in main memory is called a machine language program or (more commonly) an executable program.

Actual processors have many more machine instructions and the instructions are much more detailed.

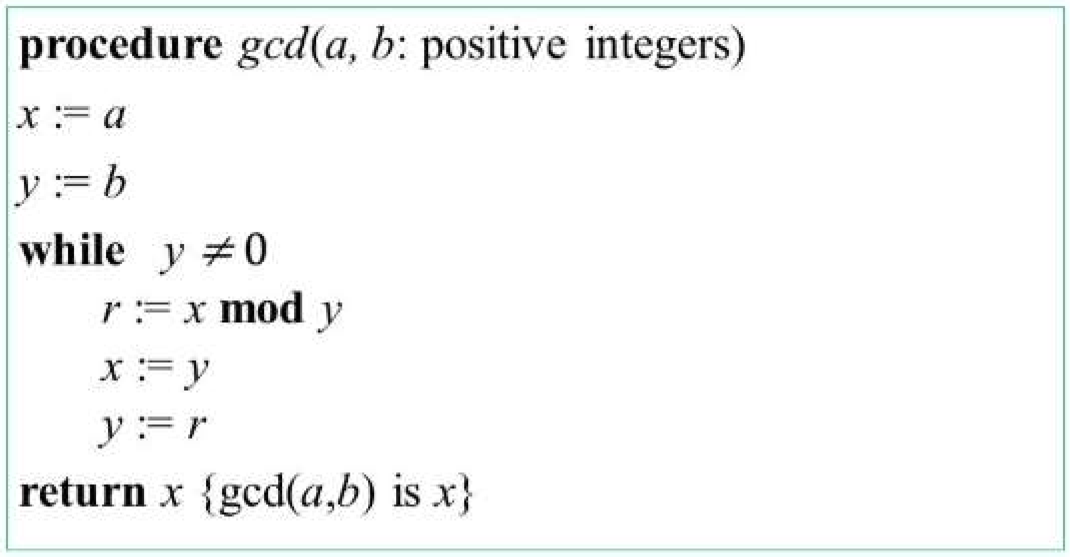
A typical processor has a thousand or more different machine instructions.

https://chortle.ccsu.edu/java5/Notes/chap04/ch04_4.html

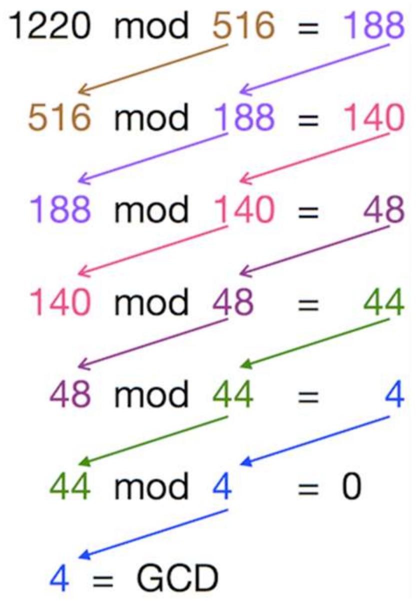
Euclid’s GCD algorithm

The Greatest Common Divisor (GCD) of two non-zero numbers is the largest positive integer that divides the number without a remainder.

Proposed by ancient Greek mathematician Euclid around 300 BC



```
procedure $gcd(a, b$: positive integers)
$x := a$
$y := b$
while $y \neq 0$
    $r := x$ mod $y$
    $x := y$
    $y := r$
return $x$ {gcd($a$,$b$) is $x$}
```



GCD Program in x86

```
55 89 e5 53  83 ec 04 83  e4 f0 e8 31  00 00 00 89  c3 e8 2a 00
00 00 39 c3  74 10 8d b6  00 00 00 00  39 c3 7e 13  29 c3 39 c3
75 f6 89 1c  24 e8 6e 00  00 00 8b 5d  fc c9 c3 29  d8 eb eb 90
```

This program calculates GCD (Greatest Common Divider) of two integers using Euclid’s algorithm.

Written in machine language expressed as hexadecimal (base 16) numbers.

Instruction set used is x86.

It can be seen that writing larger programs quickly becomes error-prone.

Assembly Languages

Were invented to allow operations to be expressed with mnemonic abbreviations.

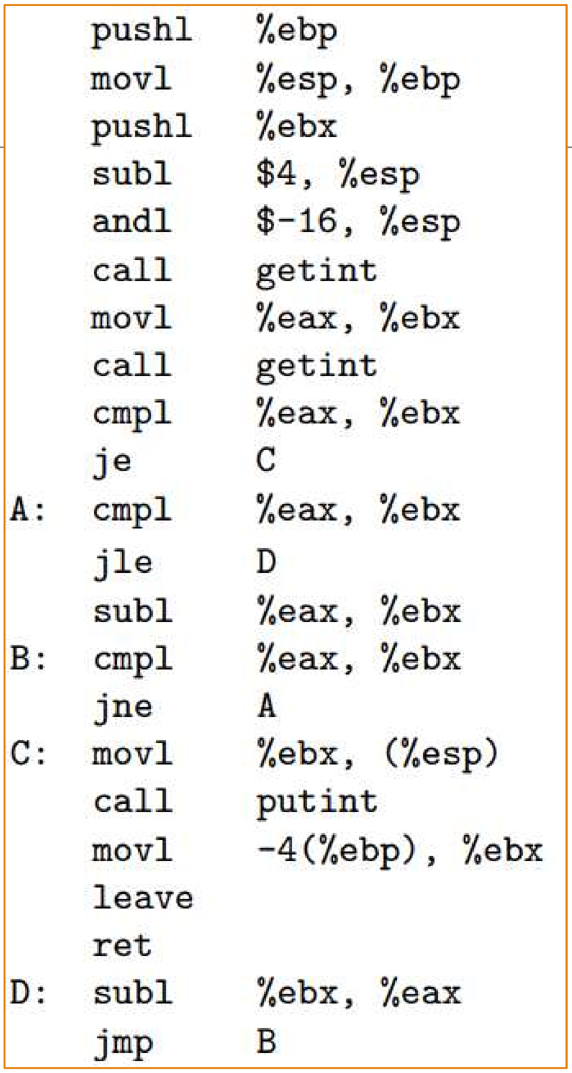
The assembly language is designed for a specific family of processors that represents various instructions in symbolic code and a more understandable form.

Assembly language is converted into executable machine code by a utility program referred to as an assembler.

Assembly Language Syntax

```
INC COUNT        ; Increment the memory variable COUNT
MOV TOTAL, 48    ; Transfer the value 48 in the memory variable TOTAL
ADD AH, BH       ; Add the content of the BH register into the AH register
AND MASK1, 128   ; Perform AND operation on the variable MASK1 and 128
ADD MARKS, 10    ; Add 10 to the variable MARKS
MOV AL, 10       ; Transfer the value 10 to the AL register
```

GCD program in Assembly Language

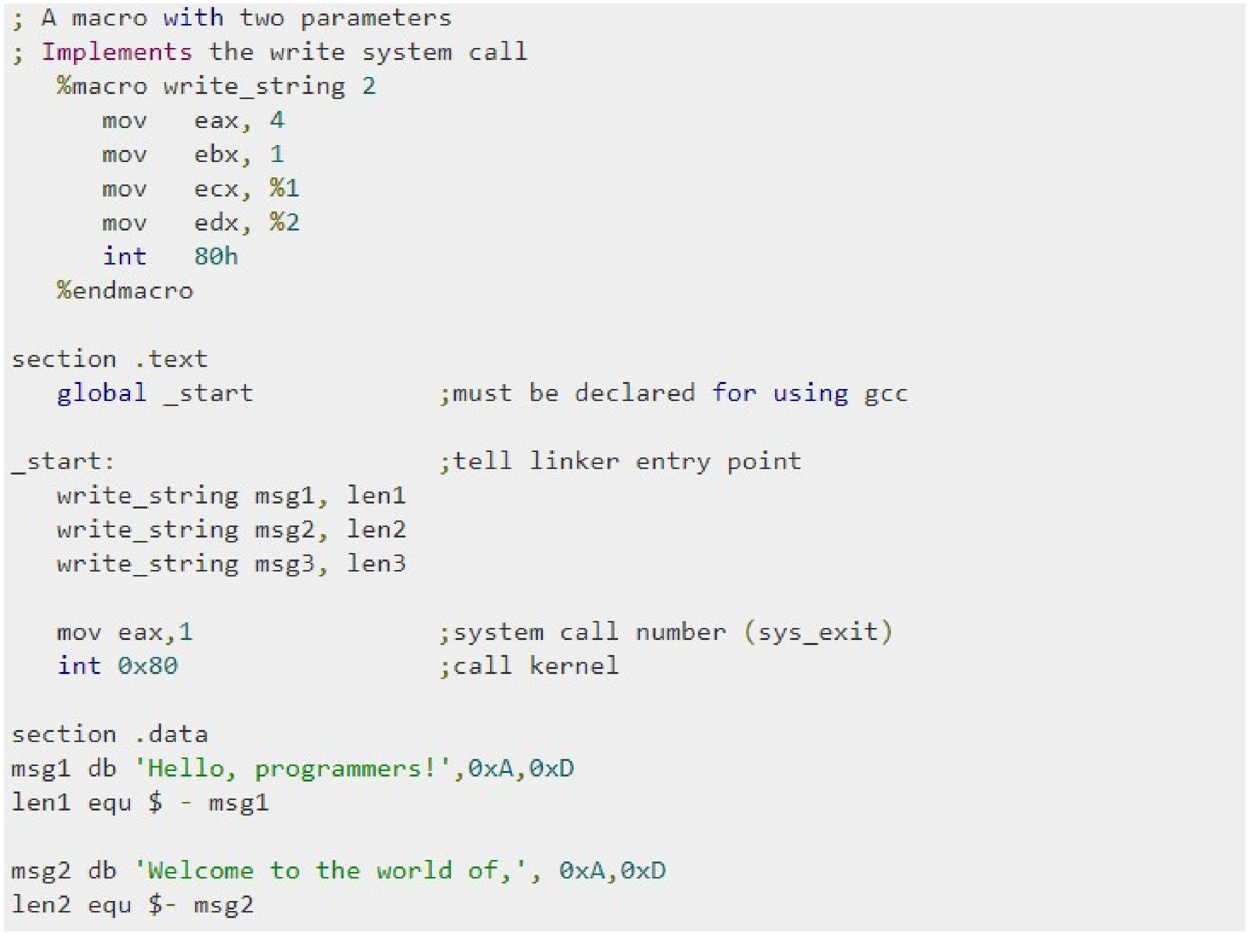


```
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %ebx
    subl    $4, %esp
    andl    $-16, %esp
    call    getint
    movl    %eax, %ebx
    call    getint
    cmpl    %eax, %ebx
    je      C
A:  cmpl    %eax, %ebx
    jle     D
    subl    %eax, %ebx
B:  cmpl    %eax, %ebx
    jne     A
C:  movl    %ebx, (%esp)
    call    putint
    movl    -4(%ebp), %ebx
    leave
    ret
D:  subl    %ebx, %eax
    jmp     B
```

x86 assembly:

The $ prefix is for constants
The % prefix is for registers

Macro Expansion in Assembly Language



```
; A macro with two parameters
; Implements the write system call
   %macro write_string 2
      mov   eax, 4
      mov   ebx, 1
      mov   ecx, %1
      mov   edx, %2
      int   80h
   %endmacro

section .text
   global _start            ;must be declared for using gcc

_start:                     ;tell linker entry point
   write_string msg1, len1
   write_string msg2, len2
   write_string msg3, len3

   mov eax,1                ;system call number (sys_exit)
   int 0x80                 ;call kernel

section .data
msg1 db 'Hello, programmers!',0xA,0xD
len1 equ $ - msg1

msg2 db 'Welcome to the world of,', 0xA,0xD
len2 equ $- msg2
```

Problems with Macro Expansion

Assemblers were eventually augmented with elaborate “macro expansion” facilities to permit programmers to define parameterized abbreviations for common sequences of instructions

Problem: each different kind of computer had to be programmed in its own assembly language

- People began to wish for a machine-independent languages

These wishes led in the mid-1950s to the development of standard higher-level languages compiled for different architectures by *compilers* which translate high-level language code to assembly or machine level language.

Compilers

Compilers are more complicated than assemblers.

One-to-one correspondence between source and target languages does not exist with high-level languages.

Initial compilers (such as Fortran compilers) were slow as human programmers could also translate code with some efforts.

Over the time, performance gap narrowed and eventually reversed.

Better hardware and improvements in compiler technology generate code better and faster compared to a human being.

Why Programming Language?

Why do we have programming languages? What is a language for?

- way of thinking -- way of expressing algorithms
- languages from the user's point of view
- abstraction of virtual machine -- way of specifying what you want
- the hardware to do without getting down into the bits
- languages from the implementor's point of view

Programming Languages

Today there are thousands of high-level programming languages, and new ones continue to emerge. Why are there so many?

- Evolution
  - E.g. goto-based control flow to while loop, case-switch statements
  - Object orientation (C++, Java), rapid development (python)
- Special Purposes
  - C is good for low level system programming
- Personal Preference
  - Terseness of C (using few words), recursive vs. iteration, pointers vs. not using pointers

Why would one code in such slow language?
People should be ashamed for coding in a non-compilable languages.
JS
This isn't even a programming language. It's ridiculous!
HTML
5
ph

Factors behind Successful Programming Languages

What makes a language successful?

- easy to learn (python, BASIC)
- easy to express things (abstraction), ease of use (C, Java)
- easy to implement (Javascript, BASIC)
- Easily available (portable copies of Pascal sent to universities)
- possible to compile to very good (fast/small) code (Fortran, C)
- Open source compiler or interpreter

Factors behind Successful Programming Languages

What makes a language successful?

- Standardization of language and libraries to ensure effective portability of code across platforms (C vs. Java)
- backing of a powerful sponsor (Java – SUN/Oracle, Ada – US Defense)
- wide dissemination at minimal cost (Java, Pascal, Turing, erlang)
- Choosing optimal language is a tradeoff
- Consider viewpoints of programmer and implementor
- Cost of implementation

Why study programming languages?

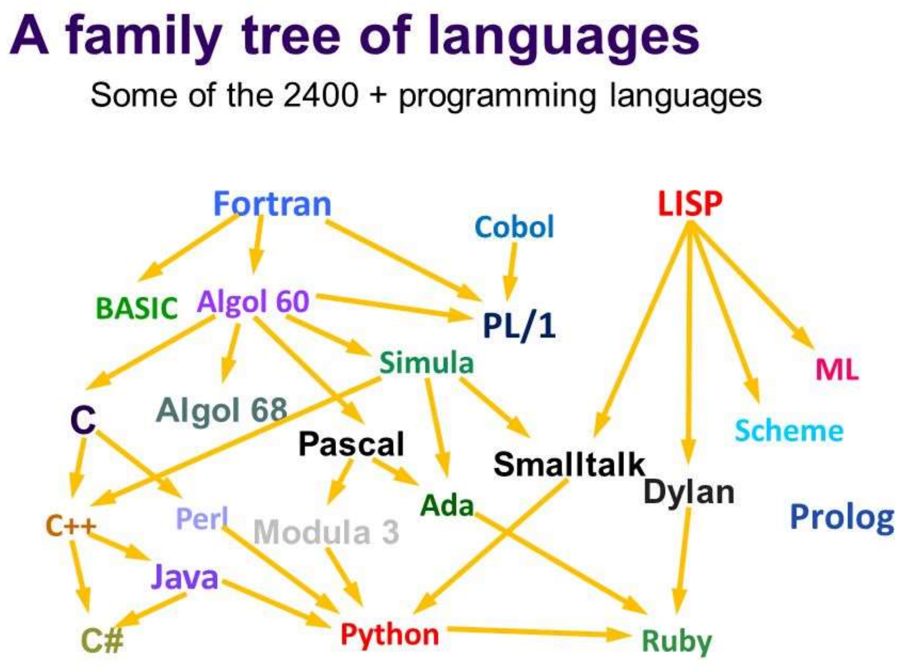
Help you choose a language:

- C vs. C++ for systems programming
- Matlab vs. Python vs. R for numerical computations
- Android vs. Java vs. Objective C vs. Javascript for embedded systems
- Python vs. Ruby vs. Scheme vs. ML for symbolic data (not purely numerical) manipulation
- Java RPC (JAX-RPC) vs. C/CORBA for networked PC programs

Why study programming languages?

Make it easier to learn new languages

- some languages are similar: easy to walk down family tree



Why study programming languages?

Concepts have even more similarity; if you think in terms of iteration, recursion, abstraction (for example), you will find it easier to assimilate the syntax and semantic details of a new language.

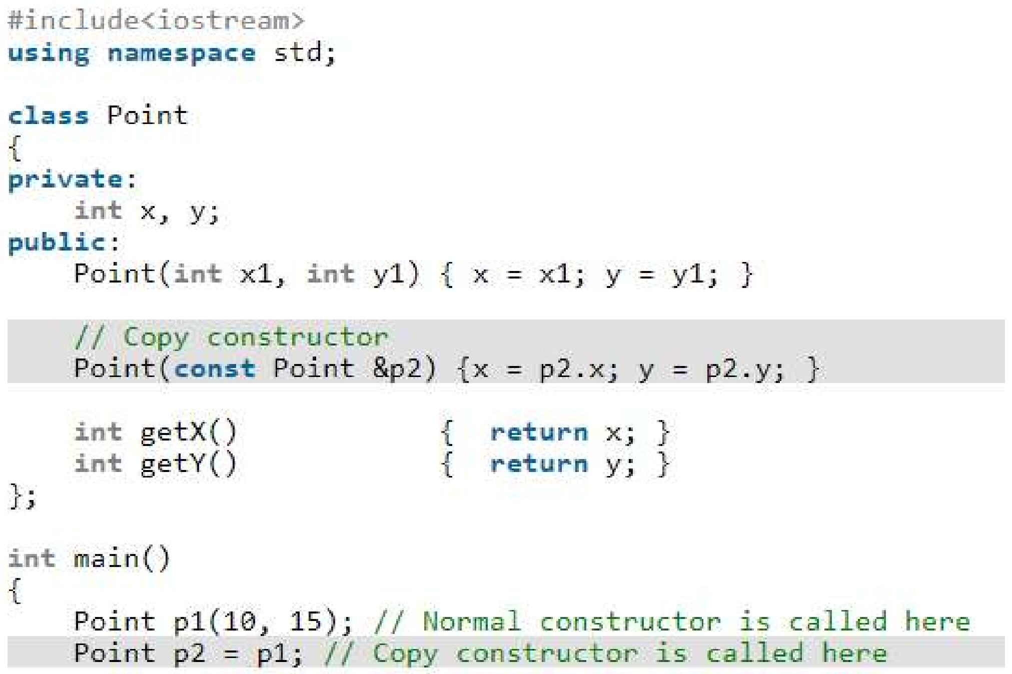
Think of an analogy to human languages: good grasp of grammar makes it easier to pick up new languages (at least Indo-European – Albanian, Armenian, Balto-Slavic, Baltic, Slavic, Celtic, Germanic).

Help making better use of obscure features:

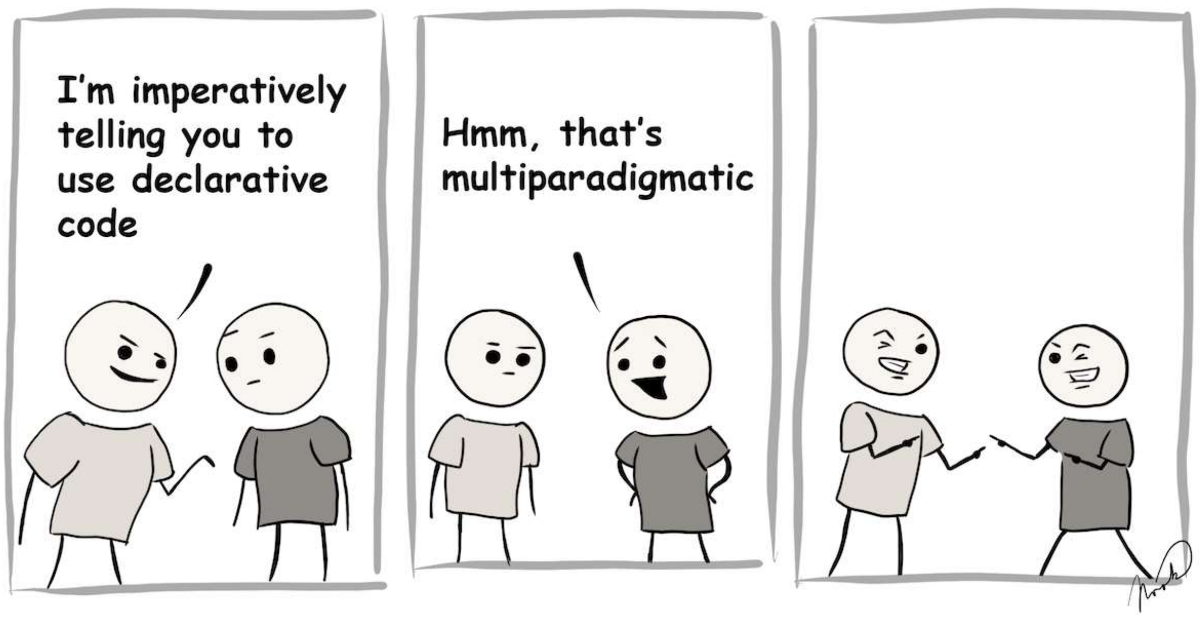
- In C, help you understand unions, arrays & pointers, separate compilation, catch and throw

Why study programming languages?

- understand implementation costs: choose between alternative ways of doing things, based on knowledge of what will be done underneath:
  - use simple arithmetic equal (use x*x instead of x**2)
  - Avoid unnecessary temporary variables and use copy constructors to minimize the cost of initialization

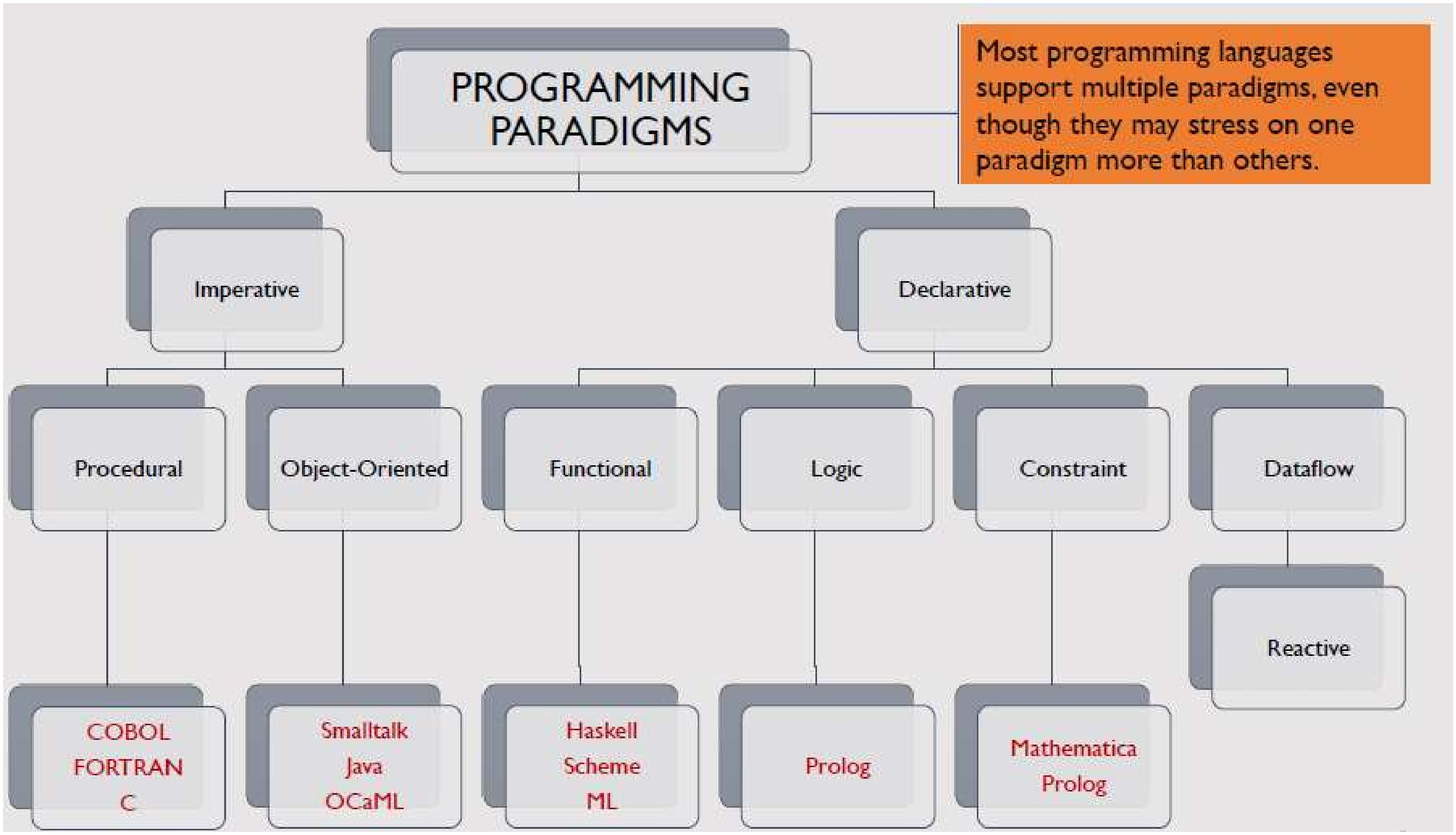


```
#include<iostream>
using namespace std;

class Point
{
private:
    int x, y;
public:
    Point(int x1, int y1) { x = x1; y = y1; }

    // Copy constructor
    Point(const Point &p2) {x = p2.x; y = p2.y; }

    int getX()            {  return x; }
    int getY()            {  return y; }
};

int main()
{
    Point p1(10, 15); // Normal constructor is called here
    Point p2 = p1; // Copy constructor is called here
```


I'm imperatively telling you to use declarative code
Hmm, that's multiparadigmatic

Imperative programming: Telling the "machine" how to do something, and as a result what you want to happen will happen. (e.g. Java code)

Declarative programming: Telling the "machine" what you would like to happen, and let the computer figure out how to do it. (e.g. HTML code, functional programming code)


PROGRAMMING PARADIGMS
Most programming languages support multiple paradigms, even though they may stress on one paradigm more than others.
Imperative
Declarative
Procedural
Object-Oriented
Functional
Logic
Constraint
Dataflow
Reactive
COBOL
FORTRAN
C
Smalltalk
Java
OCaML
Haskell
Scheme
ML
Prolog
Mathematica
Prolog

Imperative Programming

Imperative languages predominate the industry

Imperative programming describes “what” to do in terms of “how” to do it.

(Relatively) higher-level imperative languages like C are abstractions of assembly language, but they follow the same paradigm.

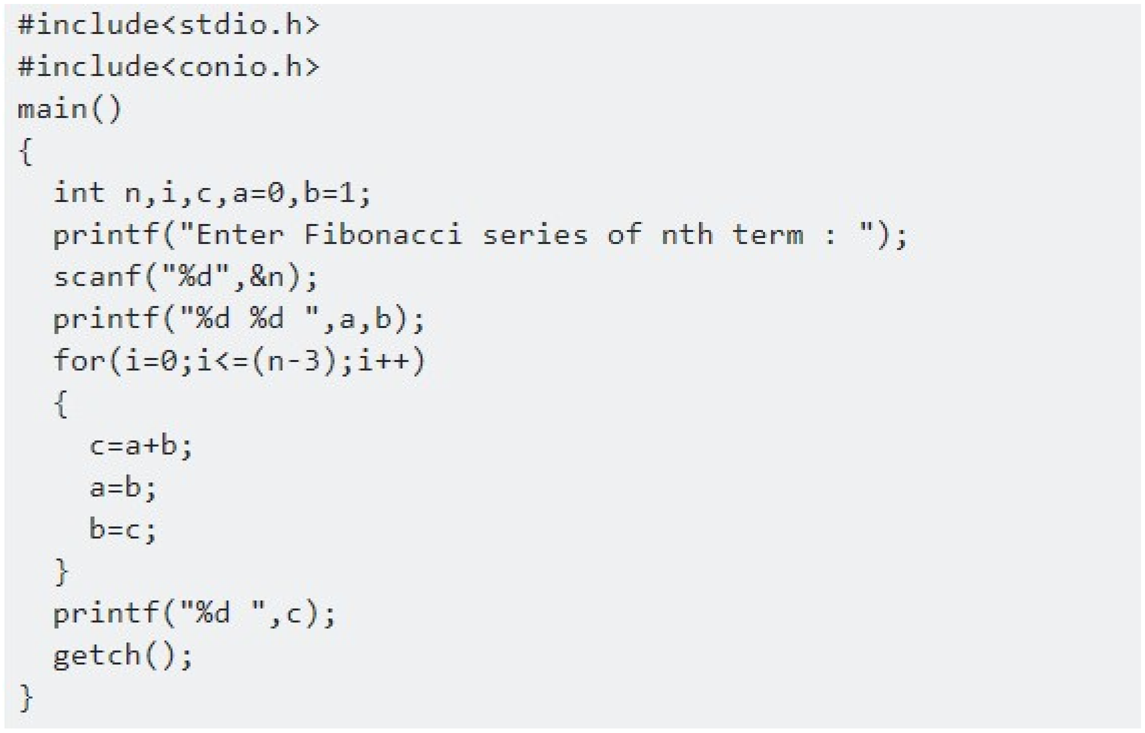
Tell the computer to perform step by step the procedure to get the final result.

Think of imperative programming as a “recipe”, where each step is an instruction about how to perform the next action, and the next action depends on the current “state” of your kitchen!

Imperative Programming

A statement is a syntactic unit of an imperative programming language that expresses some action to be carried out.

The program as a whole in such a language thus becomes a sequence of statements.



```c
#include<stdio.h>
#include<conio.h>
main()
{
  int n,i,c,a=0,b=1;
  printf("Enter Fibonacci series of nth term : ");
  scanf("%d",&n);
  printf("%d %d ",a,b);
  for(i=0;i<=(n-3);i++)
  {
    c=a+b;
    a=b;
    b=c;
  }
  printf("%d ",c);
  getch();
}
```

Procedural Programming

A type of imperative programming based on the concept of procedure calls (COBOL, Fortran, C, Pascal).

Procedures (a.k.a. routines, subroutines, or functions), simply contain a series of computational steps to be carried out.

That is, they define “how” to do what they are being asked to do. They explicitly refer to the underlying “state” (i.e., variables and their values), and are therefore within the scope of imperative programming.

Any procedure might be called at any point during a program’s execution. The call may come from other procedures or even itself.

Object Oriented Programming

A paradigm based on the concept of objects, which may contain:

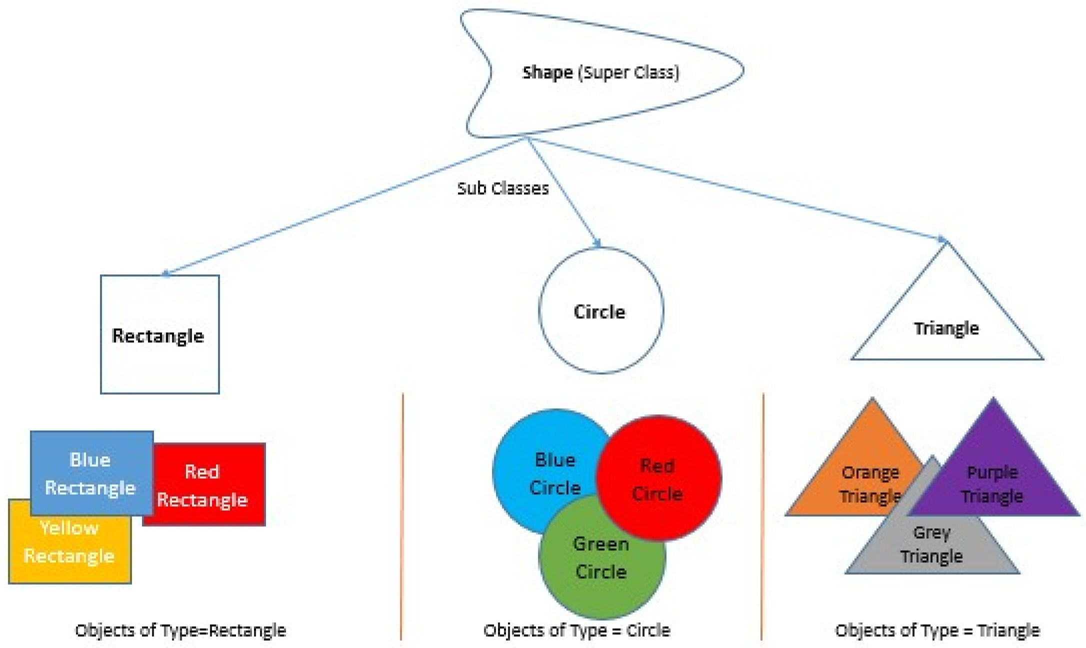
- Data in the form of fields, sometimes called attributes, and
- Code, in the form of procedures, a.k.a. methods.

An object’s procedures can access and often modify the data of the object with which they are associated (using this or self).

In OOP, programs are designed by making them out of objects that interact with one another.

Most OOP languages are class-based, i.e., objects are instances of classes (usually, this determines their type).

Object Oriented Programming



Declarative Programming

Declarative programming is about “what” to do, without specifying “how” to do it.

Just passes the input and expects the output without stating the procedure how it is done.

Of course, the computer needs to be told how to do something at some point!

But with declarative programming, those details are left to the language’s implementation.

There is a decoupling of ‘what’ and ‘how’, which makes life easier for the developer.

Kids, violence is never the answer
HTML is a Programming Language

Declarative Programming

SQL example

```
select gender, sum(income)
from income_list
group by gender;
```

HTML example

```
<!DOCTYPE html>
<html>
<body>

<div style="background-color:black;color:white;padding:20px;">
  <h2>London</h2>
  <p>London is the capital city of England. </p>
  <p>Standing on the River Thames, London has been a major
settlement for two millennia.</p>
</div>

</body>
</html>
```

Imperative vs Declarative – Metaphoric Example

Imagine you walk into your favorite coffee place and that you like to order some coffee ☕.

- The imperative approach:
- Enter the coffee shop
- Queue in the line and wait for the barista asking you for your order
- Order
- Yes, for takeaway, please
- Pay
- Present your loyalty card to collect points
- Take your order and walk out

The declarative approach:

- A large latte for takeaway, please

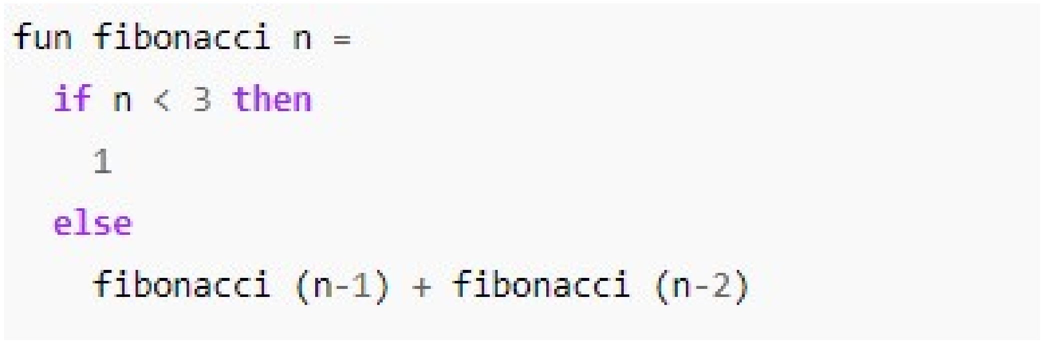
Functional Programming

Based on recursive definitions

They are inspired by a computational model called lambda calculus, developed by Alonzo Church in the 1930s.

A program is viewed as a mathematical function that transforms an input to an output. It is often defined in terms of simpler functions.

SML example – Fibonacci Series



```
fun fibonacci n =
  if n < 3 then
    1
  else
    fibonacci (n-1) + fibonacci (n-2)
```

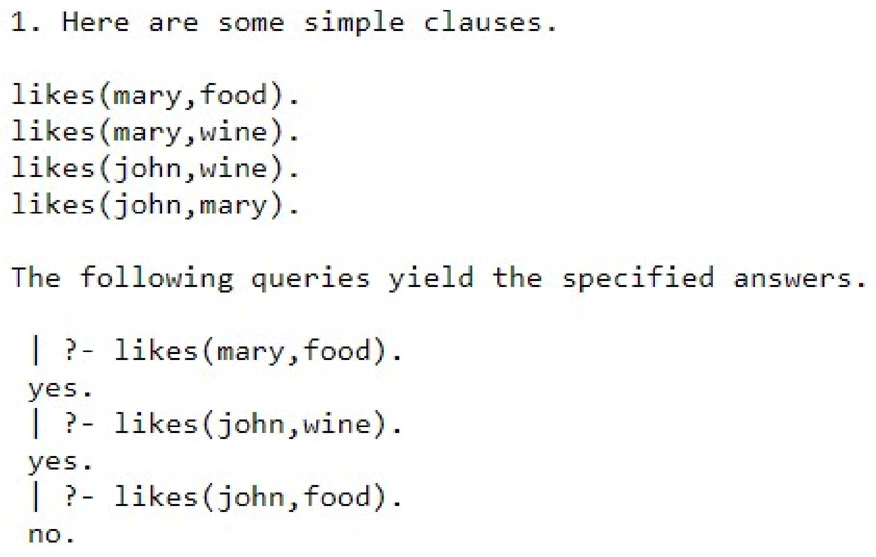
Logic/Constraint Based Programming

Based on predicate logic and an axiomatic way of finding solutions.

- The goal is often to find specific relationships that are true, starting with basic

Relations that are always true. Such a basic truth is called an axiom.

- Perhaps the best known logic programming language is Prolog.



```
1. Here are some simple clauses.

likes(mary,food).
likes(mary,wine).
likes(john,wine).
likes(john,mary).

The following queries yield the specified answers.

 | ?- likes(mary,food).
 yes.
 | ?- likes(john,wine).
 yes.
 | ?- likes(john,food).
 no.
```

Dataflow Programming

Computation is modeled as a ‘flow/stream of information’ – as a directed graph – between different operations.

- Explicitly defined input and output connect the operations.
- Each operation can be though of as a ‘black box’ function.
- In that sense, dataflow programming shares some features of functional programming.

A common example is a spread sheet program which has columns of data that are affected by other columns of data.

Reactive Programming

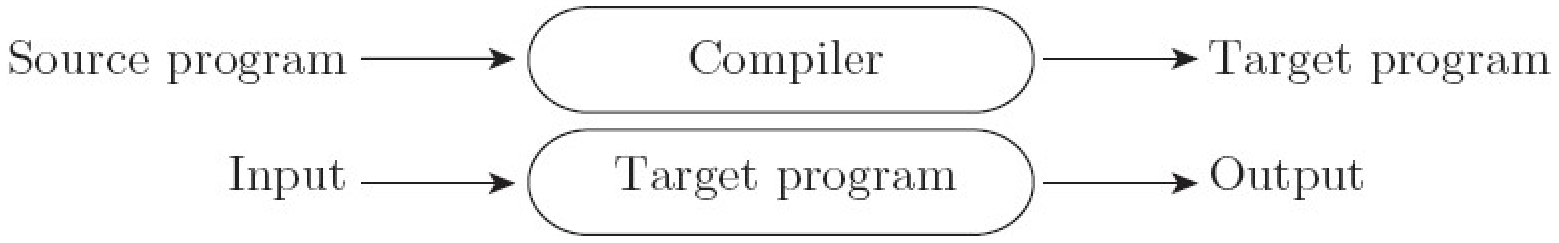
A declarative, dataflow programming paradigm where it becomes very easy to propagate changes in data.

- As an overly simplistic idea, consider a statement such as x = y + z
- In traditional imperative programming, the x is assigned the sum of the values of y and z. If the value(s) of y and/or z changes, the value of x is not affected.
- In reactive programming, the value of x is automatically updated whenever y and/or z change.

Reactive programming is extremely useful in interactive applications.

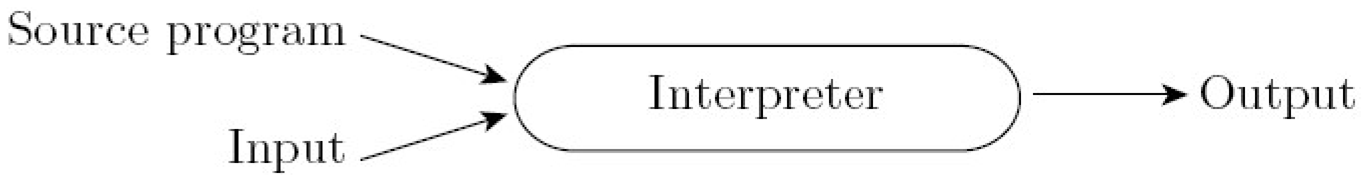
Compilation vs. Interpretation

- Compilation vs. interpretation
  - not opposites
  - not a clear-cut distinction
- Pure Compilation
  - The compiler translates the high-level source program into an equivalent target program (typically in machine language), and then goes away:



Pure Interpretation

- Interpreter stays around for the execution of the program
- Interpreter is the locus of control during execution



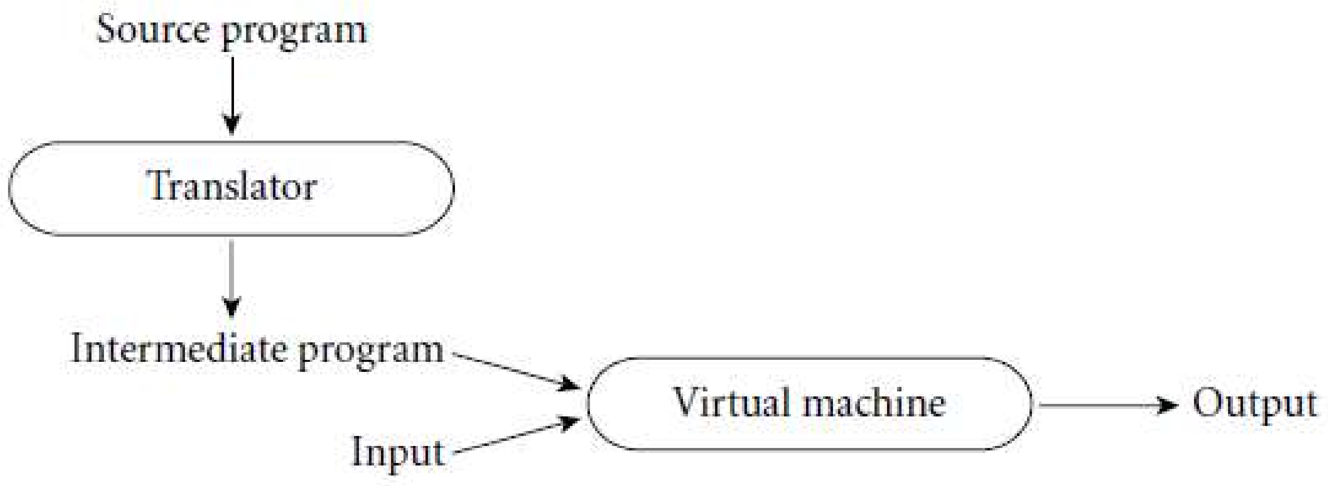
Compilation vs. Interpretation

- Interpretation:
  - Greater flexibility
  - Better diagnostics (error messages)

- Compilation
  - Better performance

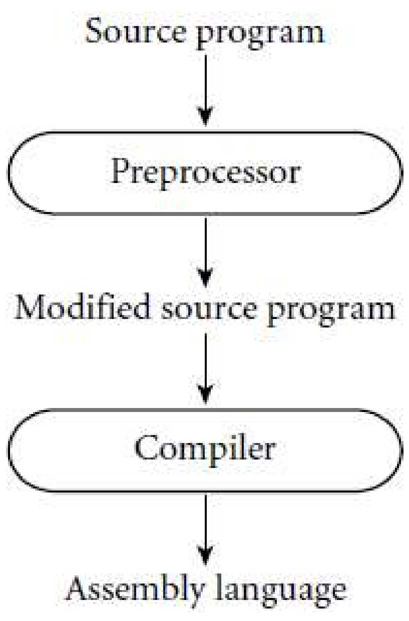
Compilation vs. Interpretation

- Common case is compilation or simple pre-processing, followed by interpretation
- Most language implementations include a mixture of both compilation and interpretation



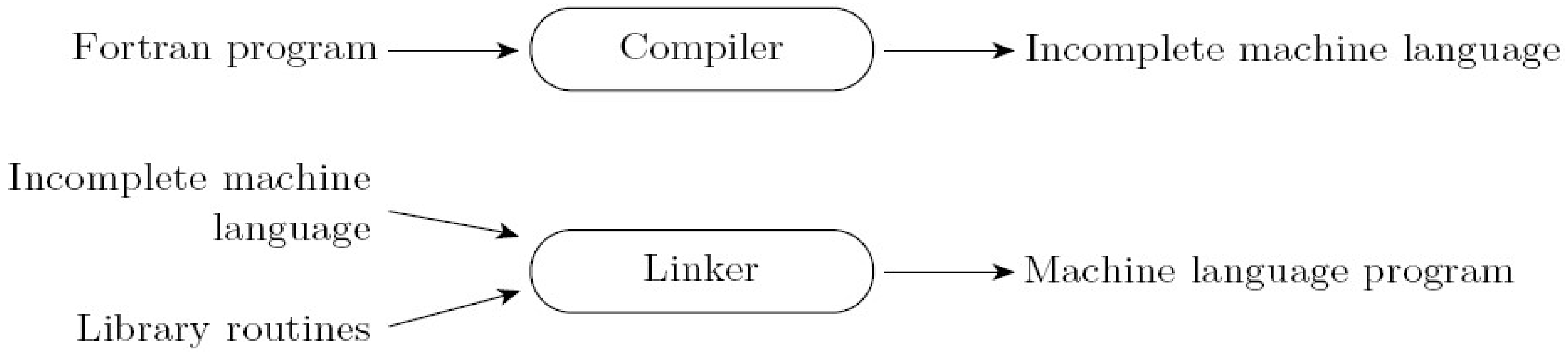
Preprocessor

- Removes comments and white space
- Groups characters into tokens (keywords, identifiers, numbers, symbols)
- Expands abbreviations in the style of a macro assembler
- Identifies higher-level syntactic structures (loops, subroutines)
- A pre-processor will often let errors through. A compiler hides further steps; a pre-processor does not
- C Preprocessor
  - Removes comments
  - Expands macros

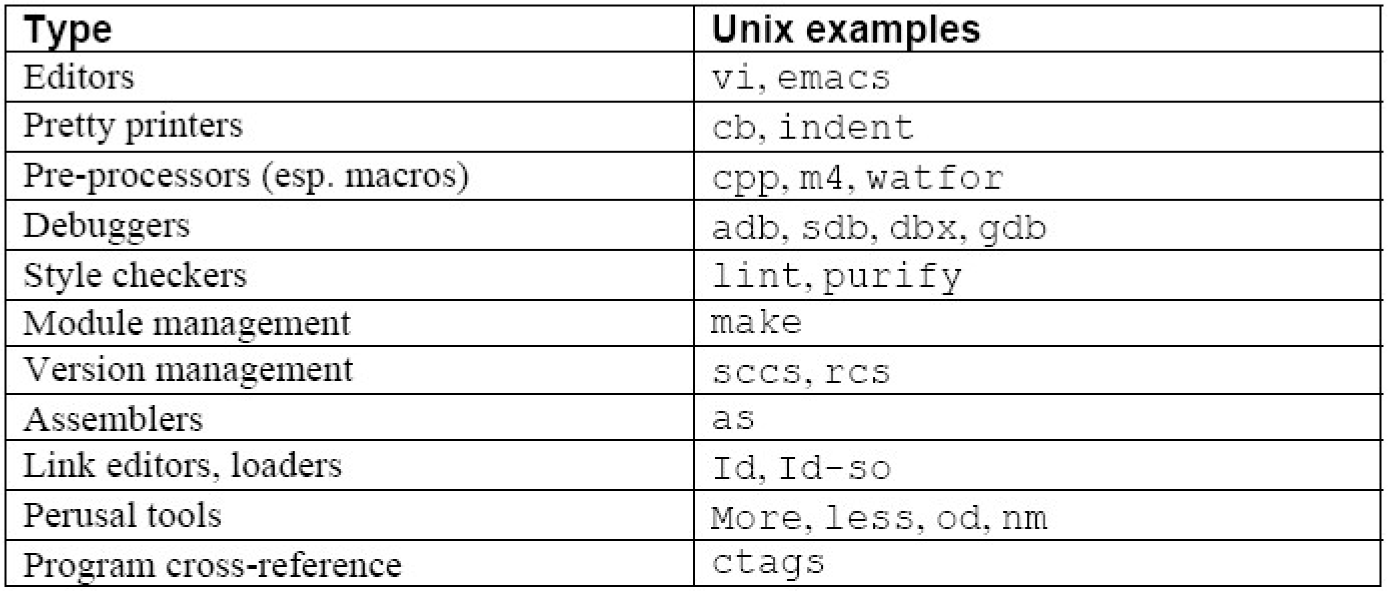


Linker

Compiler uses a *linker* program to merge the appropriate *library* of subroutines (e.g., math functions such as sin, cos, log, etc.) into the final program:



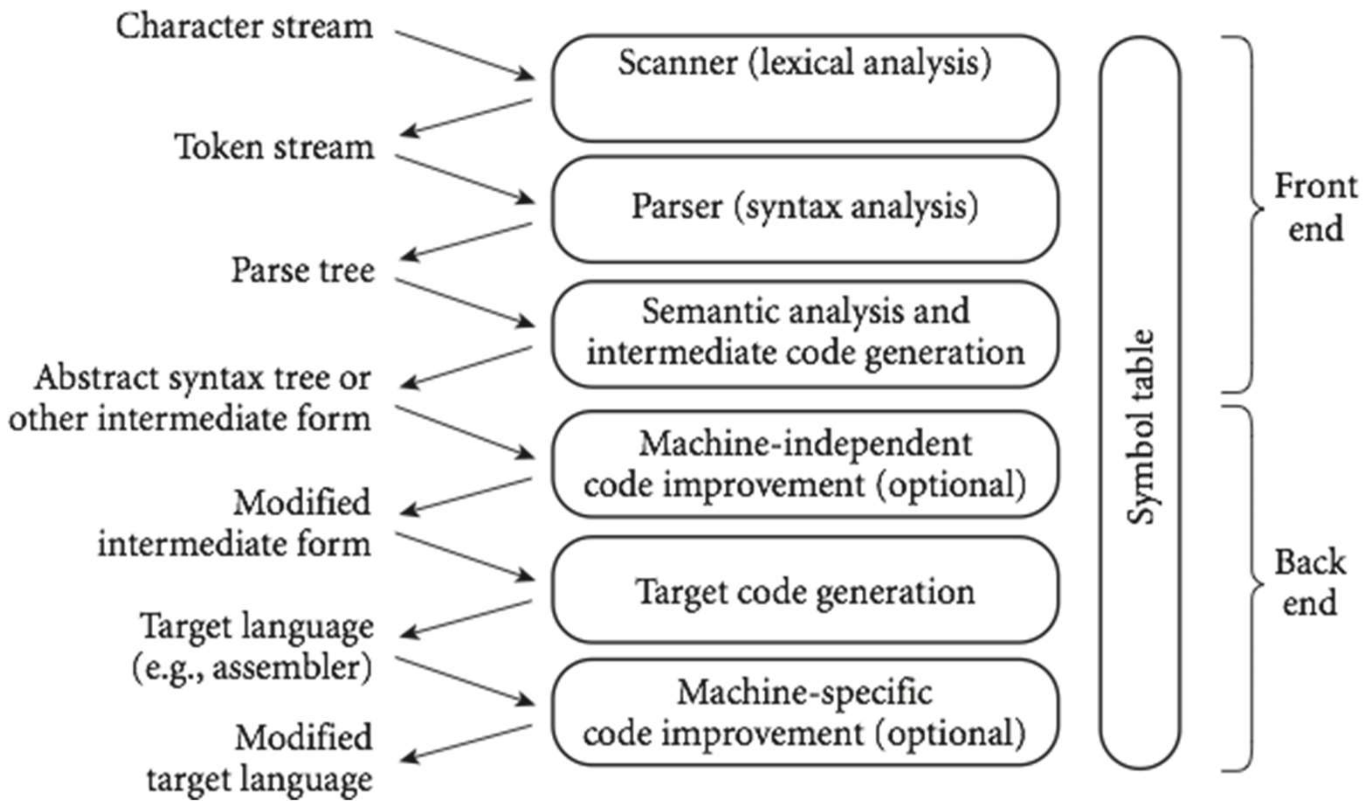
Programming Environment Tools



Type	Unix examples
Editors	`vi`, `emacs`
Pretty printers	`cb`, `indent`
Pre-processors (esp. macros)	`cpp`, `m4`, `watfor`
Debuggers	`adb`, `sdb`, `dbx`, `gdb`
Style checkers	`lint`, `purify`
Module management	`make`
Version management	`sccs`, `rcs`
Assemblers	`as`
Link editors, loaders	`Id`, `Id-so`
Perusal tools	`More`, `less`, `od`, `nm`
Program cross-reference	`ctags`

An Overview of Compilation

- Phases of Compilation



Character stream
Scanner (lexical analysis)
Token stream
Parser (syntax analysis)
Parse tree
Semantic analysis and intermediate code generation
Abstract syntax tree or other intermediate form
Machine-independent code improvement (optional)
Modified intermediate form
Target code generation
Target language (e.g., assembler)
Machine-specific code improvement (optional)
Modified target language
Symbol table
Front end
Back end

GCD Program in C

```
int main() {
int i = getint(), j = getint();
while (i != j) {
if (i > j) i = i - j;
else j = j - i;
}
putint(i);
}
```

Scanning

- divides the program into "tokens", which are the smallest meaningful units; this saves time, since character-by-character processing is slow
- we can tune the scanner better if its job is simple; it also saves complexity (lots of it) for later stages
- you can design a parser to take characters instead of tokens as input, but it isn't pretty
- scanning is recognition of a *regular language*, e.g., via DFA

GCD Program Tokens

Scanning (*lexical analysis*) and parsing recognize t he structure of the program, groups characters into *tokens*, the smallest meaningful units of the program

```
int      main   (   )        {
int      i      =   getint   (   )   ,   j   =   getint   (   )   ;
while    (      i   !=       j   )   {
if       (      i   >        j   )   i   =   i   -        j   ;
else     j      =   j        -   i   ;
}
putint   (      i   )        ;
}
```

Parsing

Parsing is recognition of a *context-free language*, e.g., via PDA

- Parsing discovers the "context free" structure of the program
- Informally, it finds the structure you can describe with syntax diagrams (the "circles and arrows" in a Pascal manual)

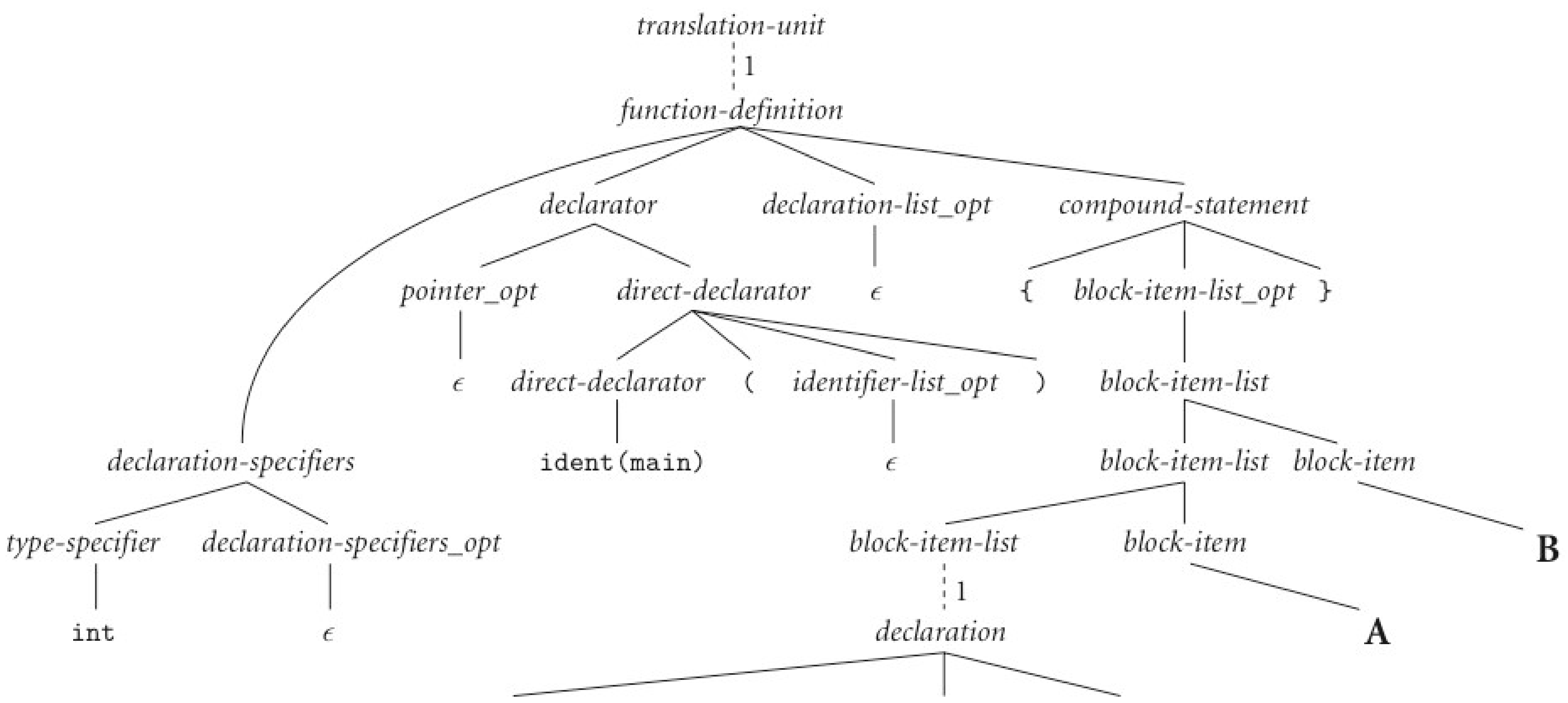
Example of Context-Free Grammar

- Example (`while` loop in C)

iteration-statement → *while (expression) statement*

statement, in turn, is often a list enclosed in braces:
statement → *compound-statement*
compound-statement → *{ block-item-list opt }*
where
block-item-list opt → *block-item-list*
or
block-item-list opt → ϵ
and
block-item-list → *block-item*
block-item-list → *block-item-list block-item*
block-item → *declaration*
block-item → *statement*

GCD Program Parse Tree

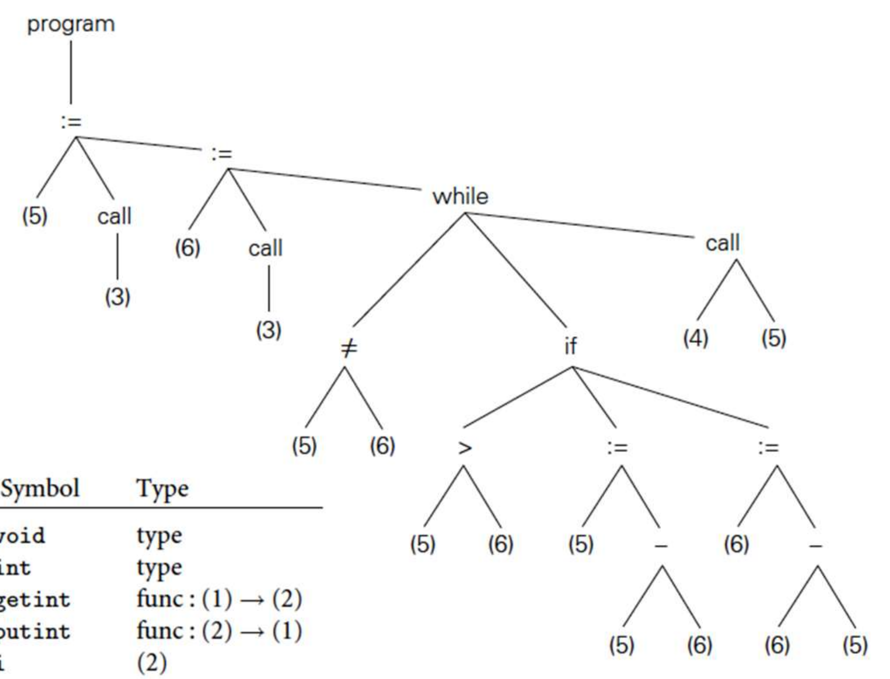


Semantic Analysis

Semantic analysis is the discovery of *meaning* in the program

- The compiler actually does what is called STATIC semantic analysis. That's the meaning that can be figured out at compile time
- Some things (e.g., array subscript out of bounds) can't be figured out until run time. Things like that are part of the program's DYNAMIC semantics

Syntax Tree and Symbol Table



Index	Symbol	Type
1	`void`	type
2	`int`	type
3	`getint`	func : (1) → (2)
4	`putint`	func : (2) → (1)
5	`i`	(2)
6	`j`	(2)

Figure 1.6 Syntax tree and symbol table for the GCD program. Note the contrast to Figure 1.5: the syntax tree retains just the essential structure of the program, omitting details that were needed only to drive the parsing algorithm.

Intermediate Form

Intermediate form (IF) done after semantic analysis (*if* the program passes all checks)

- IFs are often chosen for machine independence, ease of optimization, or compactness (these are somewhat contradictory)
- They often resemble machine code for some imaginary idealized machine; e.g. a stack machine, or a machine with arbitrarily many registers
- Many compilers actually move the code through more than one IF

Optimization and Code Generation

Optimization takes an intermediate-code program and produces another one that does the same thing faster, or in less space

- The term is a misnomer; we just *improve* code
- The optimization phase is optional

Code generation phase produces assembly language or (sometime) relocatable machine language

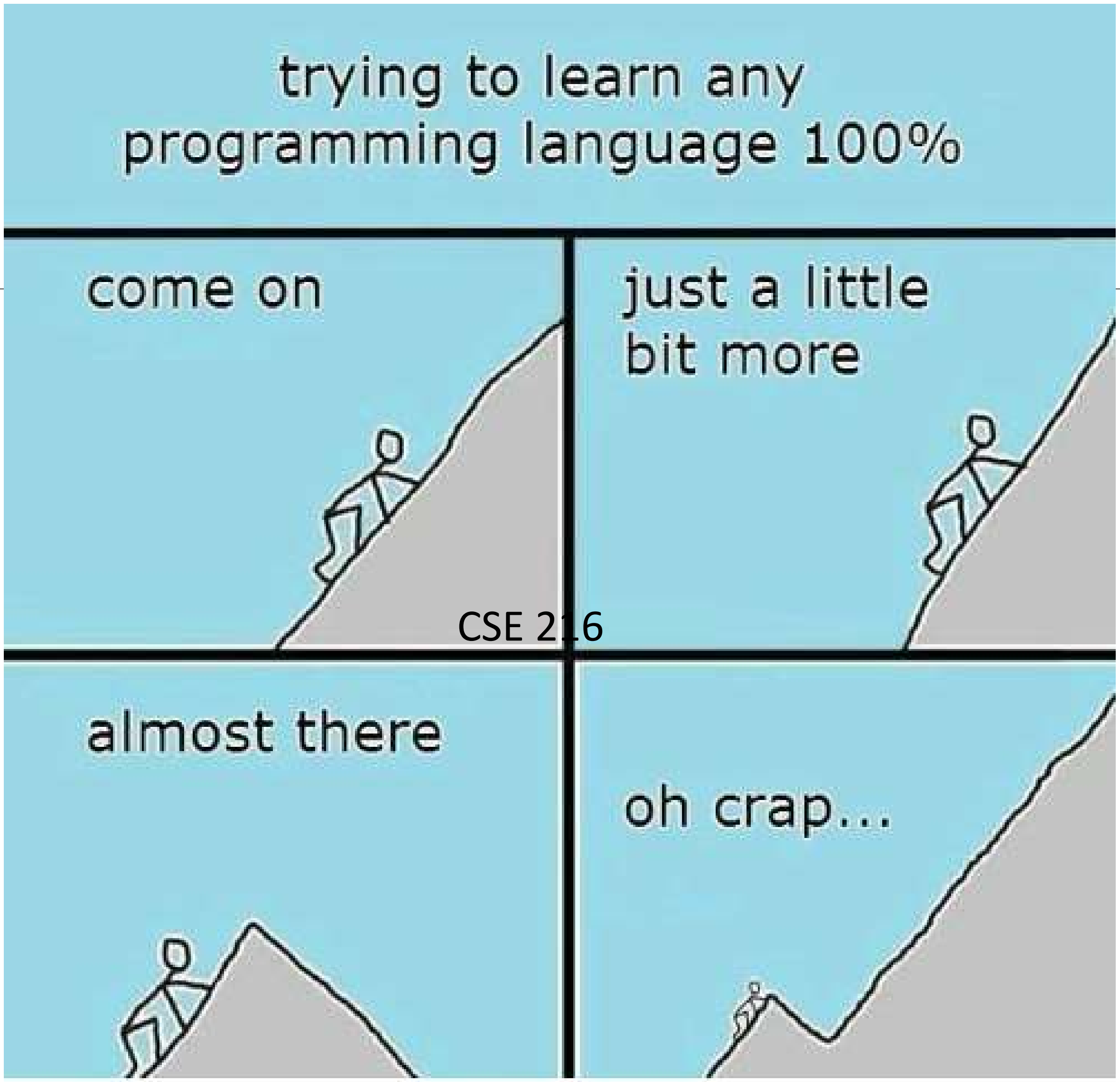
Symbol Table

All phases rely on a symbol table that keeps track of all the identifiers in the program and what the compiler knows about them

- This symbol table may be retained (in some form) for use by a debugger, even after compilation has completed

An Overview of Compilation

- Lexical and Syntax Analysis
  - Context-Free Grammar and Parsing
    - Parsing organizes tokens into a *parse tree* that represents higher-level constructs in terms of their constituents
    - Potentially recursive rules known as *context-free grammar* define the ways in which these constituents combine


trying to learn any programming language 100%
come on
just a little bit more
almost there
oh crap...

CSE 216