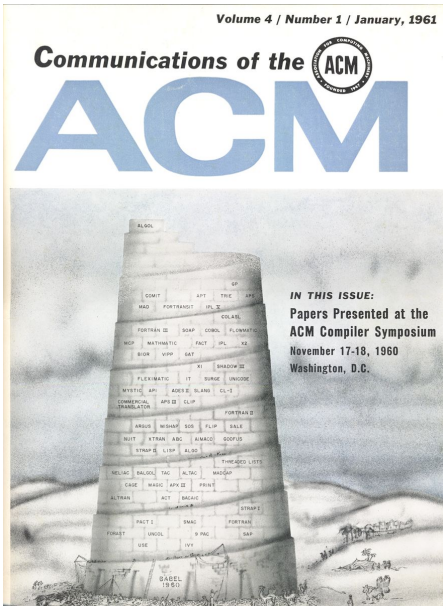
# Programming Languages

CSE 3302, Summer 2019
Module 01
Introduction




# M01

## *Introduction*

## What's a *Programming Language*?

- It's the language one uses to write a *Program*.

- OK, so what's a *Program*?
  - A set of instructions used to control the behavior of a *machine*, often a *computer*. (Wikipedia)
  - Which is kind of harsh as '*computer*' originally meant '*one* who *computes*', as in a *person*.

## Old School Computers

Astronomical calculations

Harvard College Observatory
circa 1890

# Old School Parallel Processing

Unit Record
Data Processing

circa 1920

# Old School Database

Fingerprint Records

FBI
circa 1945

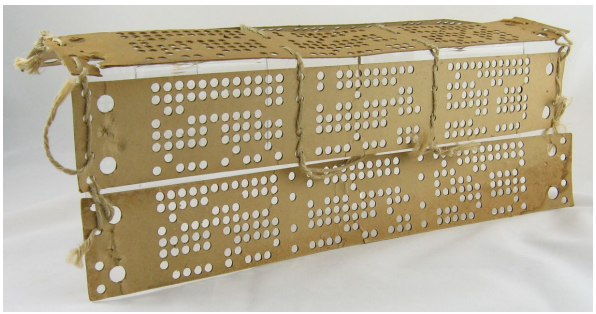
## Programming *before* Languages

- Even taking human computers out of the picture, programming existed before programming languages.
- The earliest 'computing' devices were not *stored program* devices.
- They did not embody the concept of a separate, executable set of instructions.

## Programming *before* Languages

Jacquard Loom and
'program' cards

circa 1880

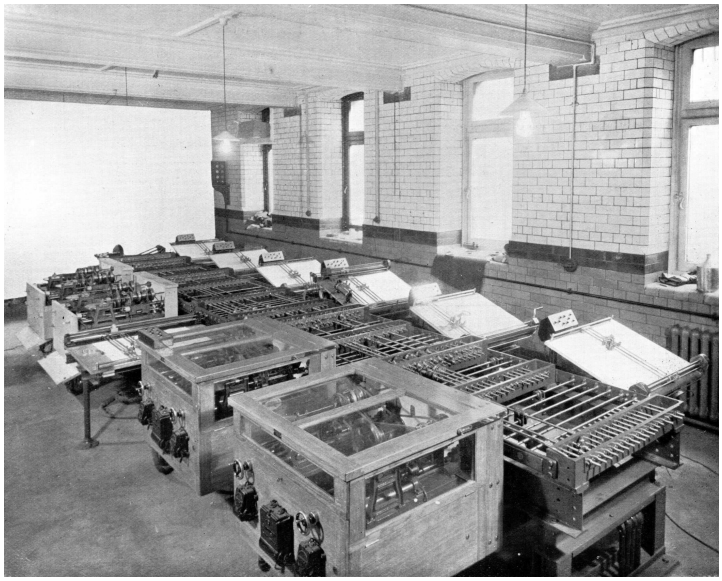


# Programming *before* Languages

Differential Analyzer

Manchester University
1938

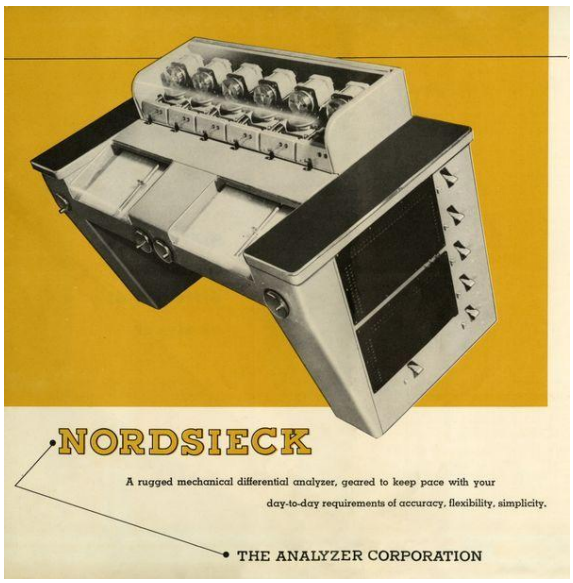
'Programmed' by
rearranging the mechanical
computation elements.



# Programming *before* Languages

Differential Analyzer

Nordsieck
University of California
Radiation Laboratory
circa 1955



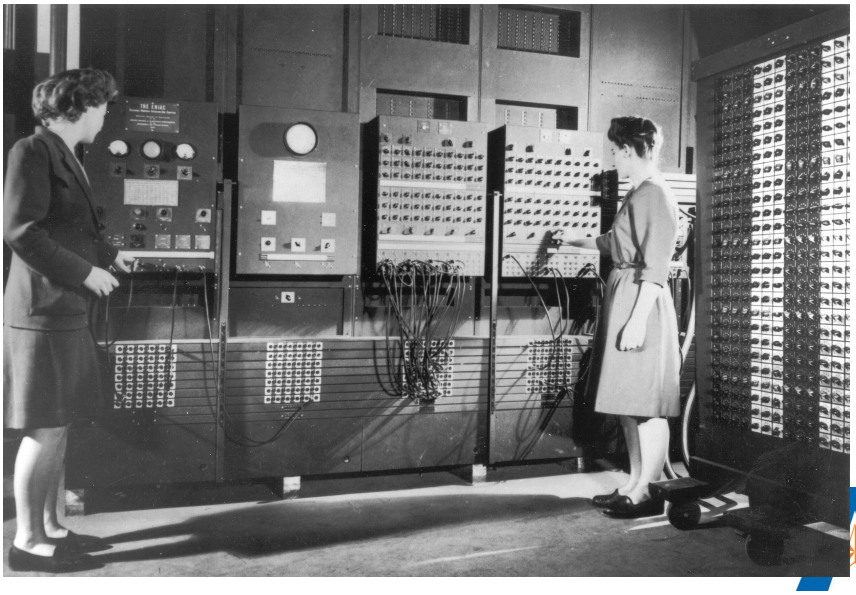

# Programming *before* Languages

ENIAC

Moore School of Engineering
University of Pennsylvania
1946

'Programmed' by rearranging cabling.

Function table set by turning knobs. Held 208 6-digit numbers, or about 520 bytes.



# Programming *before* Languages

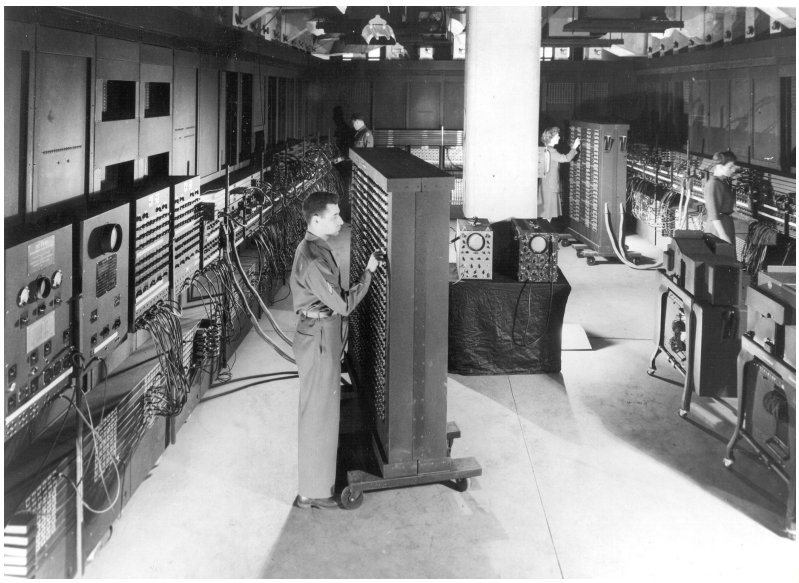
ENIAC

Moore School of Engineering
University of Pennsylvania
1946

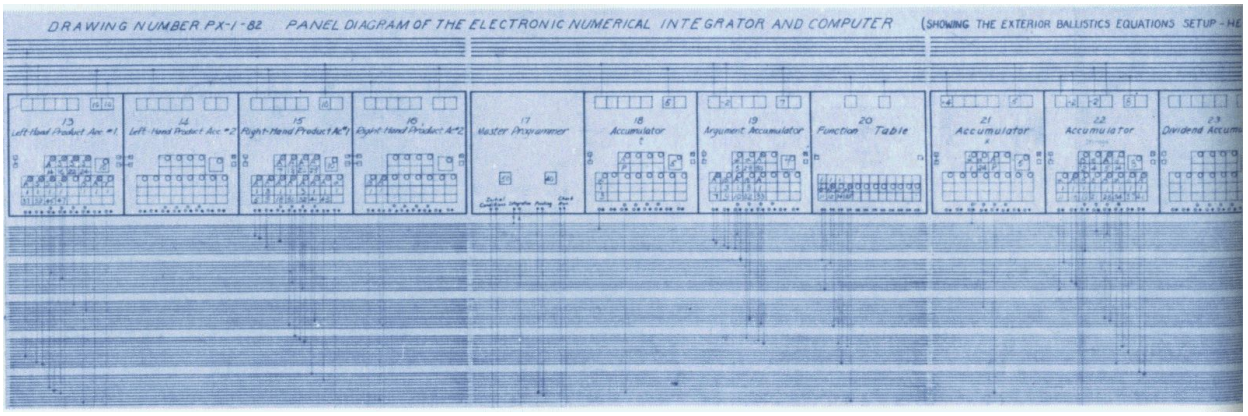
5,000 adds/s
360 multiplies/s
35 divides/s

20 k vacuum tubes
30 tons
150 kW

At first several tubes burned out each day, later reduced to one per two days.



# Programming *before* Languages



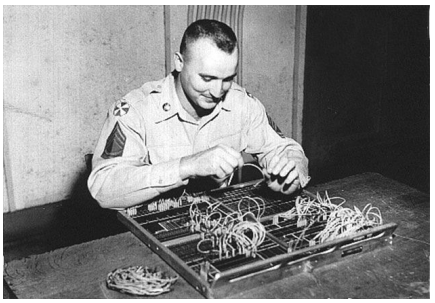

An ENIAC 'program' was a wiring diagram.

# Programming *before* Languages

IBM 407
Accounting Machine

US Army Redstone Arsenal
1961

SPC James R. Eheman
1955

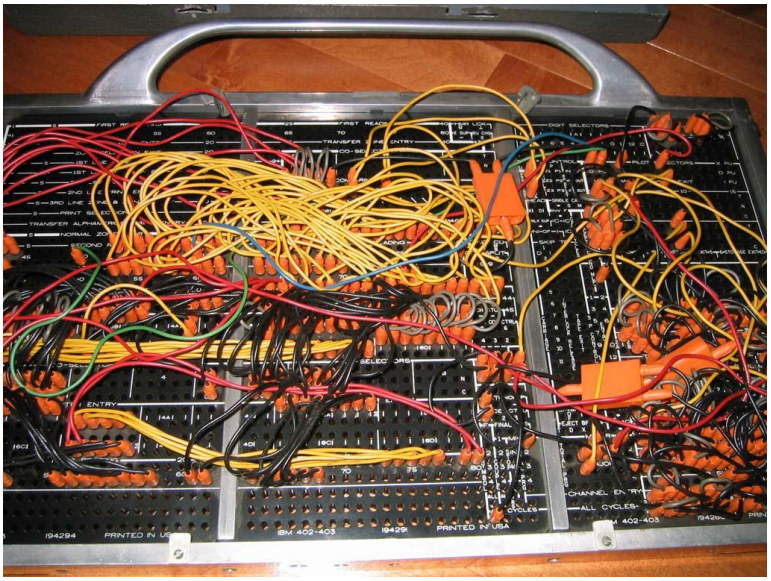


# Programming *before* Languages

IBM 402
Accounting Machine

(Chris Shrigley)
1961

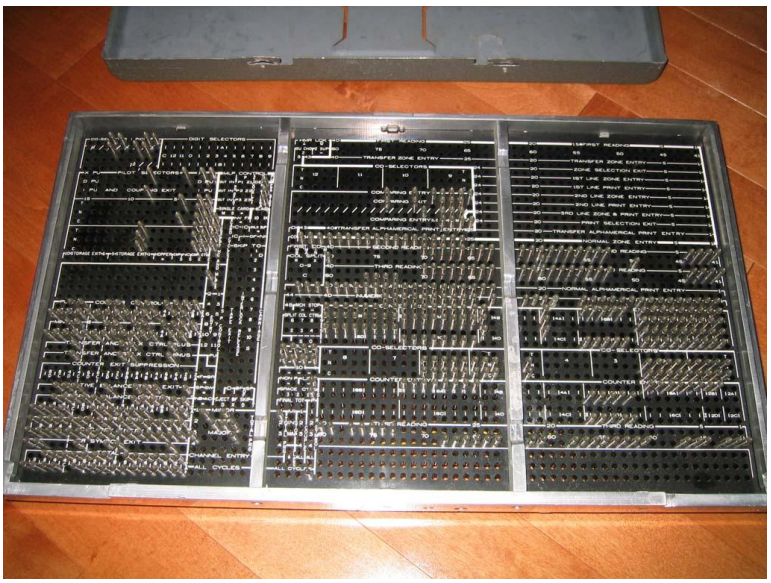
*'Profit and Loss Summary'*




# Programming *before* Languages

IBM 402

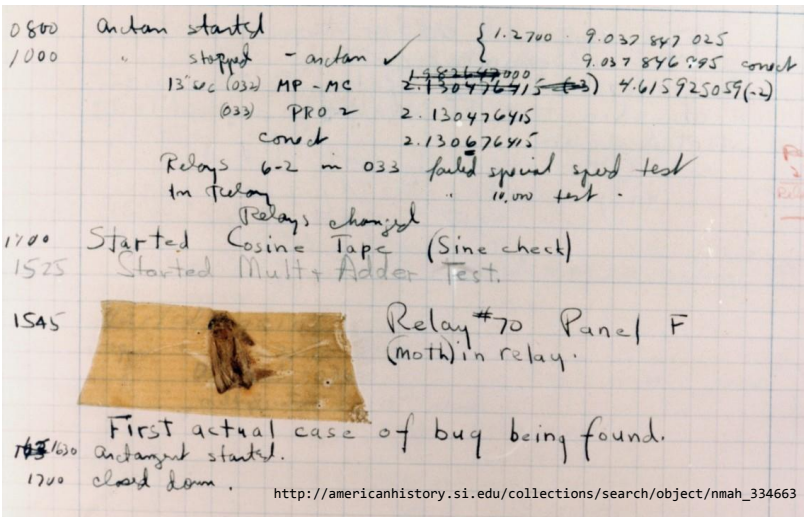
US Army Redstone Arsenal
1961



# Programming *before* Languages

- Pretty tedious, huh?
- Not very flexible and highly error prone.
- Also, the hardware itself was not very reliable and subject to the most … *interesting* … problems.

***1947 Sep 9*** LTjg Grace Hopper was working on the *Harvard University Mark II Aiken Relay Calculator*. During testing, a *moth* was found trapped between points at Relay #70, Panel F. The operators removed and affixed the moth to the computer log, with the entry "First actual case of bug being found." They put out the word that they had "debugged" the machine, thus introducing the term "debugging a computer program."




# Harvard University Mark II Aiken Relay Calculator

http://sites.harvard.edu/~chsi/markone/about.html

http://abb.com

# First FORTRAN Manual, 1956 Oct 15

*The IBM Mathematical Formula Translating System* FORTRAN *is an automatic coding system for the IBM 704 EDPM. More precisely, it is a 704 program which accepts a* source program *written in a language – the* FORTRAN *language – closely resembling the ordinary language of mathematics, and which produces an* object program *in 704 machine language, ready to be run on a 704.*

FORTRAN *therefore in effect transforms the 704 into a machine with which communication can be made in a language more concise and more familiar than the 704 language itself. The result should be a considerable reduction in the training required to program, as well as in the time consumed in writing programs and eliminating their errors.*

# Programming Languages — The Early Four

- FORTRAN (1954) ‘FORmula TRANslating system’
  - More practical alternative to assembly language.
  - Motivated theory and design of compilers.

```
C AREA OF A TRIANGLE WITH A STANDARD SQUARE ROOT FUNCTION
C INPUT - TAPE READER UNIT 5, INTEGER INPUT
C OUTPUT - LINE PRINTER UNIT 6, REAL OUTPUT
C INPUT ERROR DISPLAY ERROR OUTPUT CODE 1 IN JOB CONTROL LISTING
      READ INPUT TAPE 5, 501, IA, IB, IC
  501 FORMAT (3I5)
C IA, IB, AND IC MAY NOT BE NEGATIVE OR ZERO
C FURTHERMORE, THE SUM OF TWO SIDES OF A TRIANGLE
C MUST BE GREATER THAN THE THIRD SIDE, SO WE CHECK FOR THAT, TOO
      IF (IA) 777, 777, 701
  701 IF (IB) 777, 777, 702
  702 IF (IC) 777, 777, 703
  703 IF (IA+IB-IC) 777, 777, 704
  704 IF (IA+IC-IB) 777, 777, 705
  705 IF (IB+IC-IA) 777, 777, 799
  777 STOP 1
C USING HERON'S FORMULA WE CALCULATE THE
C AREA OF THE TRIANGLE
  799 S = FLOATF (IA + IB + IC) / 2.0
      AREA = SQRTF( S * (S - FLOATF(IA)) * (S - FLOATF(IB)) *
     +     (S - FLOATF(IC)))
      WRITE OUTPUT TAPE 6, 601, IA, IB, IC, AREA
  601 FORMAT (4H A= ,I5,5H  B= ,I5,5H  C= ,I5,8H  AREA= ,F10.2,
     +        13H SQUARE UNITS)
      STOP
      END
```

# Programming Languages — The Early Four

- ALGOL (1958) ‘ALGOrithmic Language’
  - Means of communicating numerical methods and other procedures between people and of realizing such on a variety of machines.
  - Massive contributor to the development of other languages.

```
procedure Absmax(a) Size:(n, m) Result:(y) Subscripts:(i, k);
    value n, m; array a; integer n, m, i, k; real y;
comment The absolute greatest element of the matrix a, of size n by m
    is transferred to y, and the subscripts of this element to i and k;
begin
    integer p, q;
    y := 0; i := k := 1;
    for p := 1 step 1 until n do
        for q := 1 step 1 until m do
            if abs(a[p, q]) > y then
                begin y := abs(a[p, q]);
                    i := p; k := q
                end
end Absmax
```

# Programming Languages — The Early Four

- LISP (1958) ‘LISt Processor’
  - A practical mathematical notation for computer programs, influenced by the lambda calculus.
  - Root language of AI; pioneered many computer science ideas.

```
(defun eval (e a)
 (cond
   ((atom e) (assoc e a))
   ((atom (car e))
    (cond
      ((eq (car e) 'quote) (cadr e))
      ((eq (car e) 'atom)  (atom   (eval (cadr e) a)))
      ((eq (car e) 'eq)    (eq     (eval (cadr e) a)
                                   (eval (caddr e) a)))
      ((eq (car e) 'car)   (car    (eval (cadr e) a)))
      ((eq (car e) 'cdr)   (cdr    (eval (cadr e) a)))
      ((eq (car e) 'cons)  (cons   (eval (cadr e) a)
                                   (eval (caddr e) a)))
      ((eq (car e) 'cond)  (evcon (cdr e) a))
      ('t (eval (cons (assoc (car e) a)
                      (cdr e))
                a))))
   ((eq (caar e) 'label)
    (eval (cons (caddar e) (cdr e))
          (cons (list (cadar e) (car e)) a)))
   ((eq (caar e) 'lambda)
    (eval (caddar e)
          (append (pair (cadar e) (evlis (cdr e) a))
                  a)))))
```

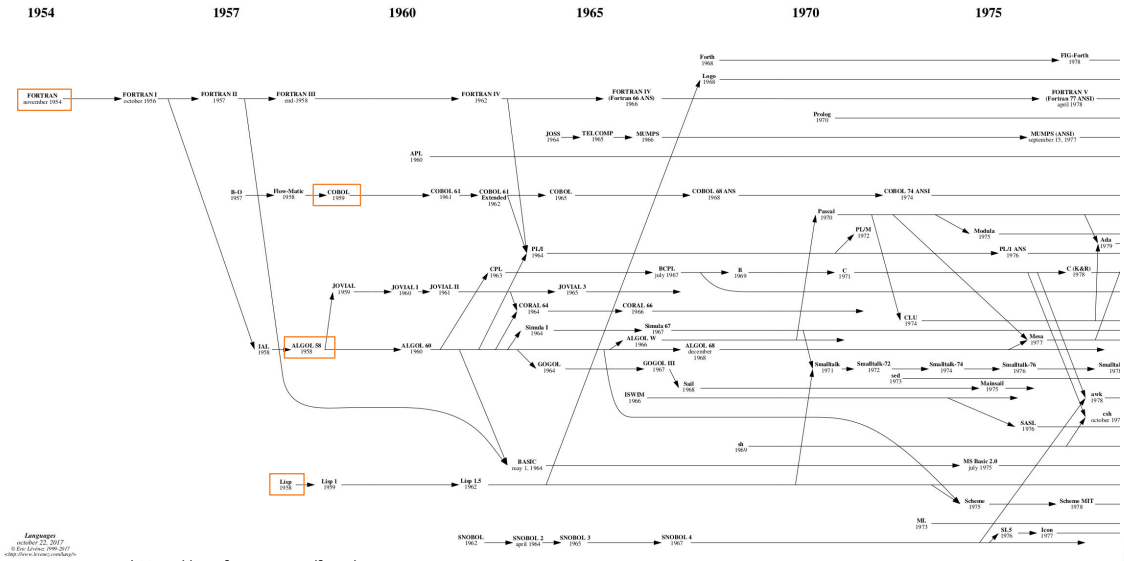
# Programming Languages — The Early Four

- COBOL (1959) ‘COmmon Business-Oriented Language’
  - US Department of Defense attempt to create a portable data processing programming language.
  - Designed to be self-documenting and readable by non-programmers.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.       SAMPLE.
AUTHOR.           J.P.E. HODGSON.
DATE-WRITTEN.     4 February 2000

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT STUDENT-FILE     ASSIGN TO SYSIN
        ORGANIZATION IS LINE SEQUENTIAL.
    SELECT PRINT-FILE       ASSIGN TO SYSOUT
        ORGANIZATION IS LINE SEQUENTIAL.

DATA DIVISION.
FILE SECTION.
FD  STUDENT-FILE
    RECORD CONTAINS 43 CHARACTERS
    DATA RECORD IS STUDENT-IN.
01  STUDENT-IN               PIC X(43).

FD  PRINT-FILE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS PRINT-LINE.
01  PRINT-LINE               PIC X(80).

WORKING-STORAGE SECTION.
01  DATA-REMAINS-SWITCH  PIC X(2)   VALUE SPACES.
01  RECORDS-WRITTEN      PIC 99.

01  DETAIL-LINE.
    05  FILLER           PIC X(7)   VALUE SPACES.
    05  RECORD-IMAGE     PIC X(43).
    05  FILLER           PIC X(30)  VALUE SPACES.

01  SUMMARY-LINE.
    05  FILLER           PIC X(7)   VALUE SPACES.
    05  TOTAL-READ       PIC 99.
    05  FILLER           PIC X      VALUE SPACE.
    05  FILLER           PIC X(17)
                VALUE  'Records were read'.
    05  FILLER           PIC X(53)  VALUE SPACES.
```

```
PROCEDURE DIVISION.

PREPARE-SENIOR-REPORT.
    OPEN INPUT  STUDENT-FILE OUTPUT PRINT-FILE.
    MOVE ZERO TO RECORDS-WRITTEN.
    READ STUDENT-FILE
        AT END MOVE 'NO' TO DATA-REMAINS-SWITCH
    END-READ.
    PERFORM PROCESS-RECORDS
        UNTIL DATA-REMAINS-SWITCH = 'NO'.
    PERFORM PRINT-SUMMARY.
    CLOSE STUDENT-FILE PRINT-FILE.
    STOP RUN.

PROCESS-RECORDS.
    MOVE STUDENT-IN TO RECORD-IMAGE.
    MOVE DETAIL-LINE TO PRINT-LINE.
    WRITE PRINT-LINE.
    ADD 1 TO RECORDS-WRITTEN.
    READ STUDENT-FILE
        AT END MOVE 'NO' TO DATA-REMAINS-SWITCH
    END-READ.

PRINT-SUMMARY.
    MOVE RECORDS-WRITTEN TO TOTAL-READ.
    MOVE SUMMARY-LINE TO PRINT-LINE.
    WRITE PRINT-LINE.
```

# Programming Languages Timeline — Early Years




https://www.levenez.com/lang/

# Early Four to … *Thousands!*

- Wikipedia's *List of Programming Languages* page currently has ***721*** entries.
  - This includes only those languages thought *significant*.
  - And they left out all the various dialects of BASIC, esoteric languages, and markup languages.

- How to make any sense out of this?
  - Maybe we should just consider the *popular* languages?

https://en.wikipedia.org/wiki/List_of_programming_languages

# What's *Popular*?

https://spectrum.ieee.org/computing/software/the-2018-top-programming-languages

- How to even judge?
- The IEEE uses eleven data sources for its annual ranking.

| | | | |
|---|---|---|---|
| **Google**<br># sites indexed | **Google**<br># queries made | **Reddit**<br># posts mentioning | **Hacker News**<br># posts mentioning |
| **Github**<br># active repositories | **Github**<br># newly created | **Career Builder**<br># recent job ads | **Dice**†<br># recent job ads |
| **Stack Overflow**<br># questions mentioning | **Stack Overflow**<br># responses to questions | **Twitter**<br># tweets mentioning | **IEEE Xplore**<br># journal articles |

†Dice turned off its API in 2018. The previous year ratings include Dice info.


# What's *Popular*?

https://spectrum.ieee.org/computing/software/the-2018-top-programming-languages

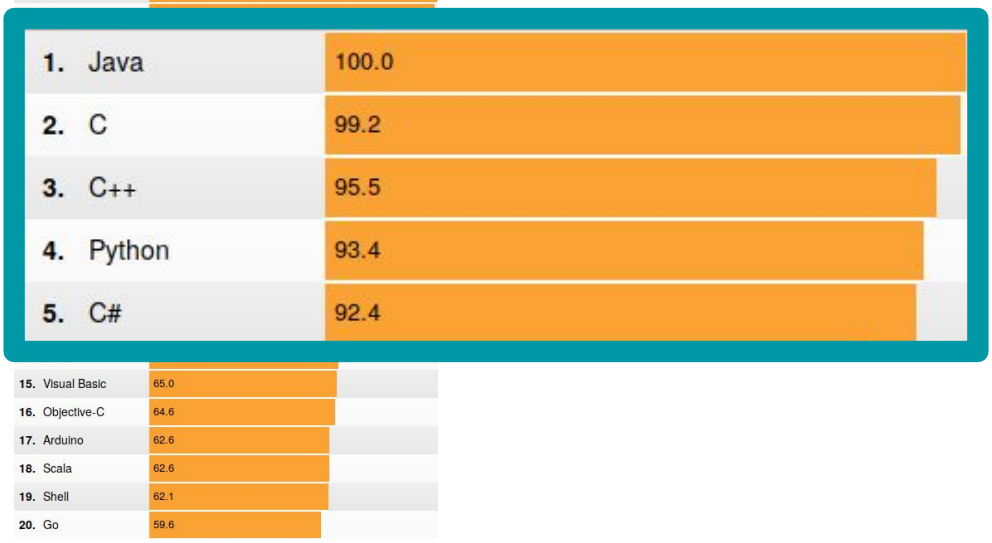

**2014**

| | | |
|---|---|---|
| 1. | Java | 100.0 |
| 2. | C | 99.2 |
| 3. | C++ | 95.5 |
| 4. | Python | 93.4 |
| 5. | C# | 92.4 |
| 6. | PHP | 84.6 |
| 7. | JavaScript | 84.3 |
| 8. | Ruby | 78.7 |
| 9. | R | 74.7 |
| 10. | Matlab | 73.3 |
| 11. | Assembly | 70.7 |
| 12. | Perl | 70.2 |
| 13. | HTML | 68.2 |
| 14. | SQL | 65.8 |
| 15. | Visual Basic | 65.0 |
| 16. | Objective-C | 64.6 |
| 17. | Arduino | 62.6 |
| 18. | Scala | 62.6 |
| 19. | Shell | 62.1 |
| 20. | Go | 59.6 |

# What's *Popular*?

https://spectrum.ieee.org/computing/software/the-2018-top-programming-languages

**2014**




| | | |
|---|---|---|
| 15. | Visual Basic | 65.0 |
| 16. | Objective-C | 64.6 |
| 17. | Arduino | 62.6 |
| 18. | Scala | 62.6 |
| 19. | Shell | 62.1 |
| 20. | Go | 59.6 |


# What's *Popular*?

https://spectrum.ieee.org/computing/software/the-2018-top-programming-languages

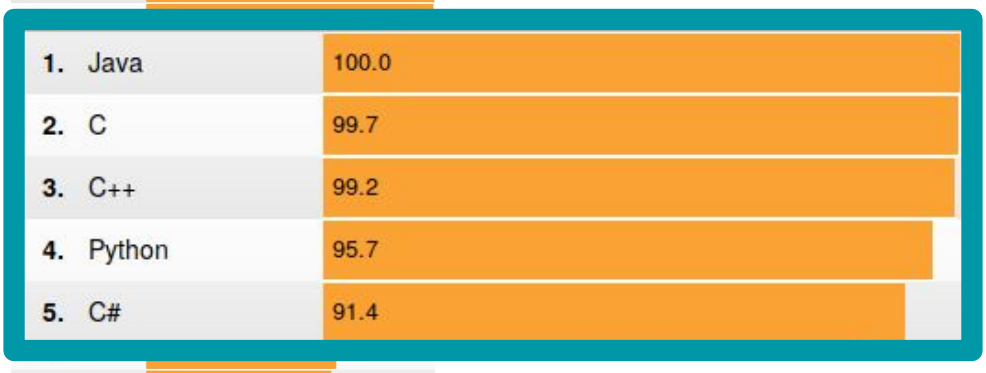

**2015**

| | | |
|---|---|---|
| 1. | Java | 100.0 |
| 2. | C | 99.7 |
| 3. | C++ | 99.2 |
| 4. | Python | 95.7 |
| 5. | C# | 91.4 |
| 6. | PHP | 84.2 |
| 7. | R | 83.6 |
| 8. | JavaScript | 82.7 |
| 9. | Ruby | 74.8 |
| 10. | Matlab | 73.6 |
| 11. | Shell | 70.4 |
| 12. | Assembly | 69.3 |
| 13. | Go | 66.8 |
| 14. | Perl | 66.0 |
| 15. | Visual Basic | 64.2 |
| 16. | Arduino | 63.8 |
| 17. | Scala | 61.3 |
| 18. | Swift | 61.3 |
| 19. | HTML | 58.1 |
| 20. | Objective-C | 57.3 |

# What's *Popular*?

https://spectrum.ieee.org/computing/software/the-2018-top-programming-languages

**2015**




| | | |
|---|---|---|
| 15. | Visual Basic | 64.2 |
| 16. | Arduino | 63.8 |
| 17. | Scala | 61.3 |
| 18. | Swift | 61.3 |
| 19. | HTML | 58.1 |
| 20. | Objective-C | 57.3 |

# What's *Popular*?

https://spectrum.ieee.org/computing/software/the-2018-top-programming-languages

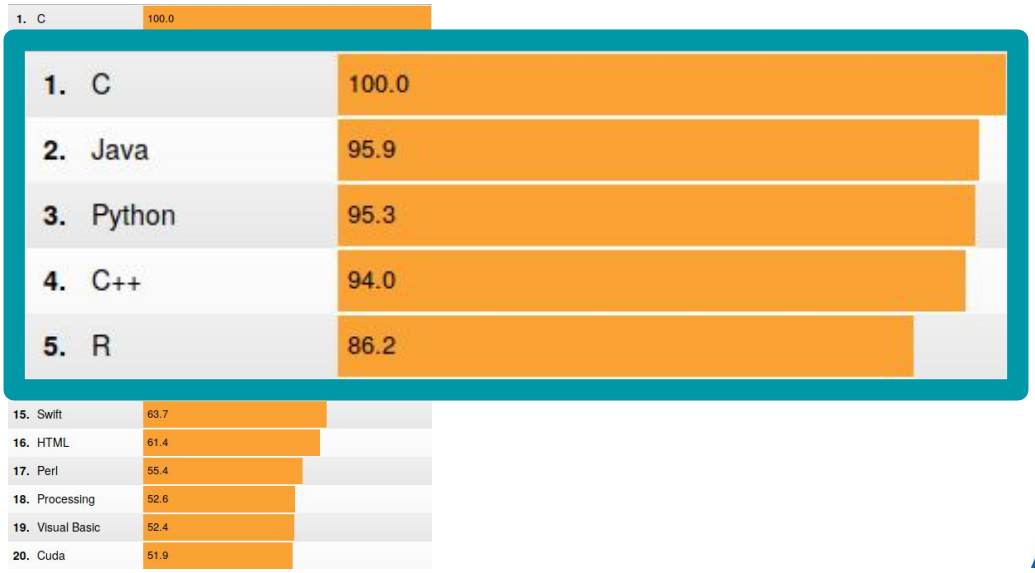
**2016**

| | | |
|---|---|---|
| 1. | C | 100.0 |
| 2. | Java | 95.9 |
| 3. | Python | 95.3 |
| 4. | C++ | 94.0 |
| 5. | R | 86.2 |
| 6. | C# | 83.3 |
| 7. | PHP | 79.2 |
| 8. | JavaScript | 79.0 |
| 9. | Ruby | 70.8 |
| 10. | Go | 70.3 |
| 11. | Arduino | 69.5 |
| 12. | Assembly | 67.4 |
| 13. | Matlab | 67.2 |
| 14. | Scala | 64.0 |
| 15. | Swift | 63.7 |
| 16. | HTML | 61.4 |
| 17. | Perl | 55.4 |
| 18. | Processing | 52.6 |
| 19. | Visual Basic | 52.4 |
| 20. | Cuda | 51.9 |

https://spectrum.ieee.org/computing/software/the-2018-top-programming-languages

# What's *Popular*?

**2016**




| | | |
|---|---|---|
| 15. | Swift | 63.7 |
| 16. | HTML | 61.4 |
| 17. | Perl | 55.4 |
| 18. | Processing | 52.6 |
| 19. | Visual Basic | 52.4 |
| 20. | Cuda | 51.9 |

https://spectrum.ieee.org/computing/software/the-2018-top-programming-languages

# What's *Popular*?

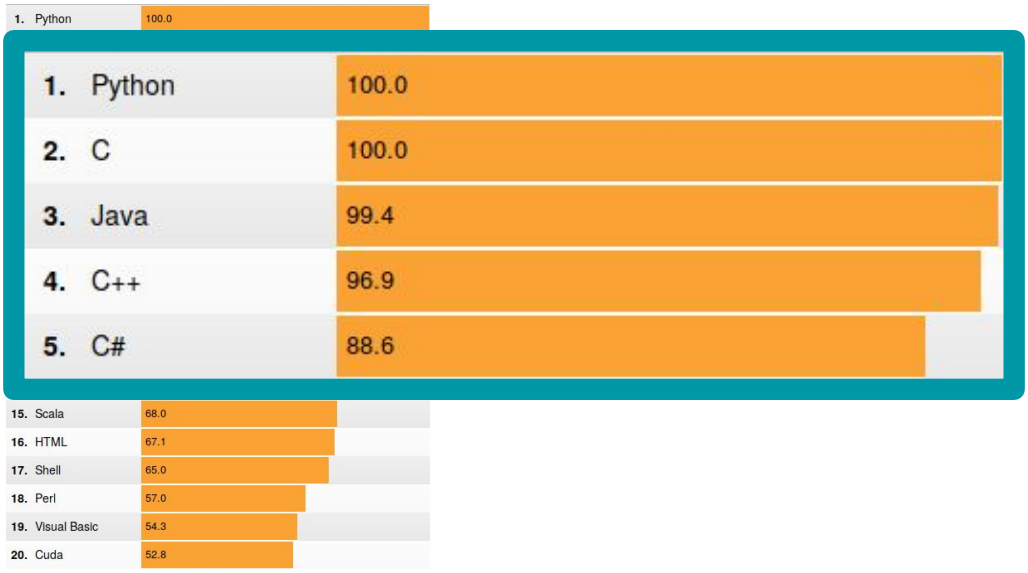
**2017**

| | | |
|---|---|---|
| 1. | Python | 100.0 |
| 2. | C | 100.0 |
| 3. | Java | 99.4 |
| 4. | C++ | 96.9 |
| 5. | C# | 88.6 |
| 6. | R | 88.1 |
| 7. | JavaScript | 85.3 |
| 8. | PHP | 81.1 |
| 9. | Go | 75.7 |
| 10. | Swift | 74.3 |
| 11. | Arduino | 72.4 |
| 12. | Ruby | 72.0 |
| 13. | Assembly | 71.7 |
| 14. | Matlab | 68.6 |
| 15. | Scala | 68.0 |
| 16. | HTML | 67.1 |
| 17. | Shell | 65.0 |
| 18. | Perl | 57.0 |
| 19. | Visual Basic | 54.3 |
| 20. | Cuda | 52.8 |

# What's *Popular*?

https://spectrum.ieee.org/computing/software/the-2018-top-programming-languages

**2017**




| | | |
|---|---|---|
| 15. | Scala | 68.0 |
| 16. | HTML | 67.1 |
| 17. | Shell | 65.0 |
| 18. | Perl | 57.0 |
| 19. | Visual Basic | 54.3 |
| 20. | Cuda | 52.8 |


# What's *Popular*?

https://spectrum.ieee.org/computing/software/the-2018-top-programming-languages


**2018**

| | | |
|---|---|---|
| 1. | Python | 100.0 |
| 2. | C++ | 99.7 |
| 3. | Java | 97.5 |
| 4. | C | 96.7 |
| 5. | C# | 89.4 |
| 6. | PHP | 84.9 |
| 7. | R | 82.9 |
| 8. | JavaScript | 82.6 |
| 9. | Go | 76.4 |
| 10. | Assembly | 74.1 |
| 11. | Matlab | 72.8 |
| 12. | Scala | 72.1 |
| 13. | Ruby | 71.4 |
| 14. | HTML | 71.2 |
| 15. | Arduino | 69.0 |
| 16. | Shell | 66.1 |
| 17. | Perl | 57.4 |
| 18. | Swift | 53.9 |
| 19. | Processing | 53.1 |
| 20. | Objective-C | 50.5 |

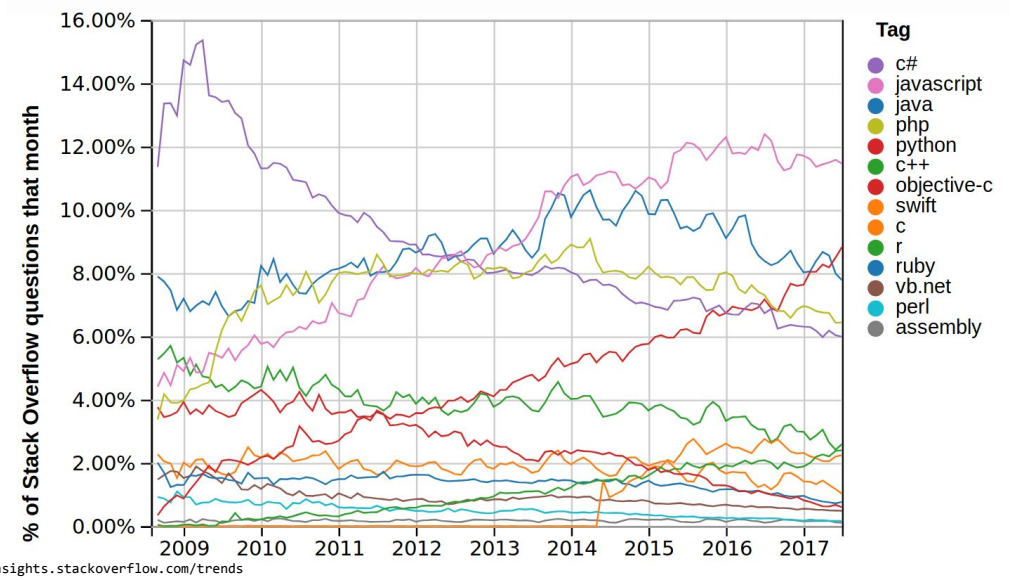
# What's *Popular*?

https://spectrum.ieee.org/computing/software/the-2018-top-programming-languages

**2018**

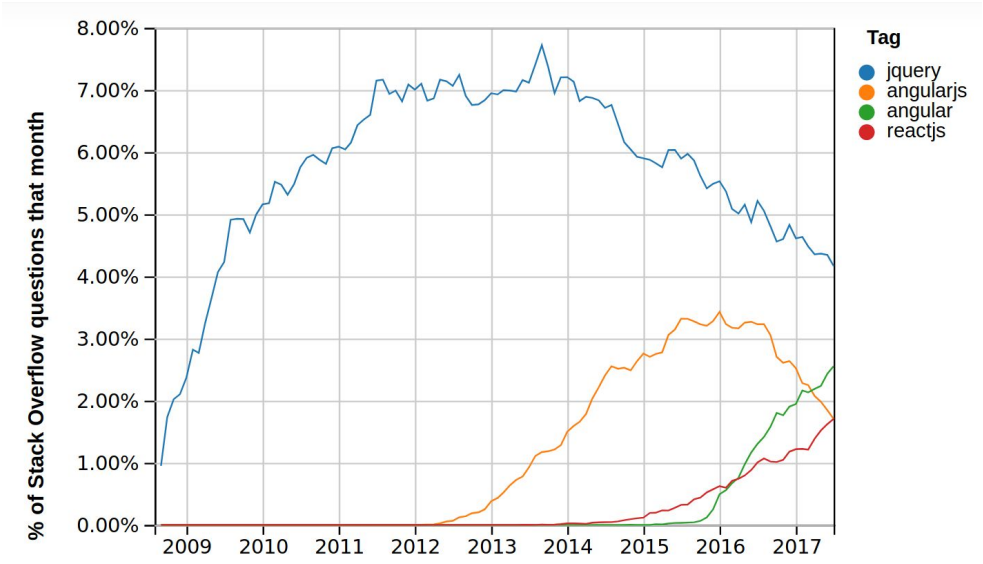
| | |
|---|---|
| 1. Python | 100.0 |
| 2. C++ | 99.7 |
| 3. Java | 97.5 |
| 4. C | 96.7 |
| 5. C# | 89.4 |
| 15. Arduino | 69.0 |
| 16. Shell | 66.1 |
| 17. Perl | 57.4 |
| 18. Swift | 53.9 |
| 19. Processing | 53.1 |
| 20. Objective-C | 50.5 |

# Stack Overflow's Perceptions (1)



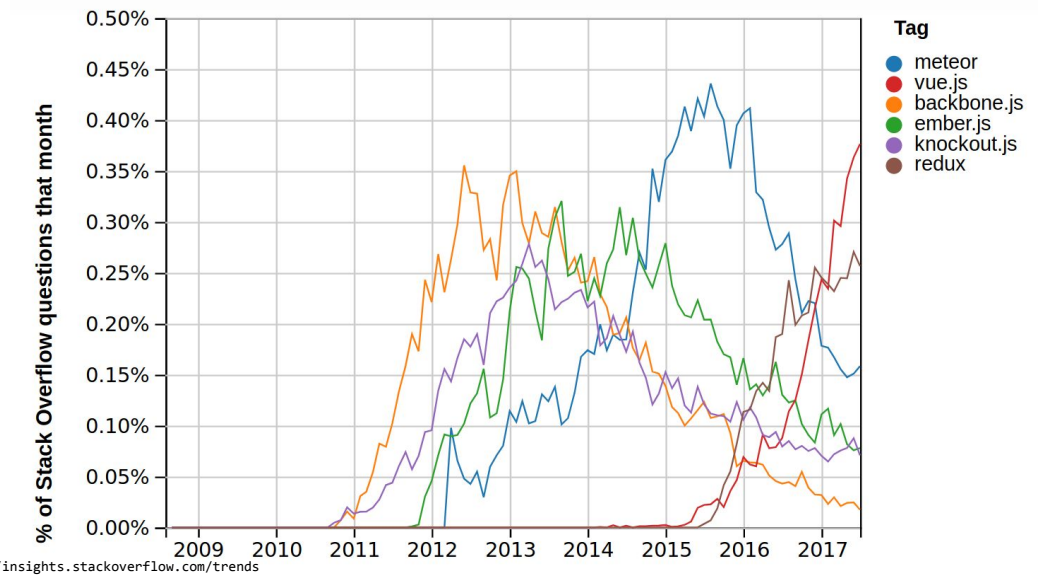

https://insights.stackoverflow.com/trends

# Stack Overflow's Perceptions (2)




https://insights.stackoverflow.com/trends

# Stack Overflow's Perceptions (3)




https://insights.stackoverflow.com/trends

## Conclusions?

- Technologies come and go
- Be prepared for whatever you like best to fall out of favor

## The Art of Language Design

- Why are there so many programming languages?
  - Remember, Wikipedia lists ***721*** "significant" languages without even including variants of BASIC, esoteric languages, or markup languages.
- Programming languages are created by humans and …
  - Depending on how one defines "language" and how one distinguishes "language" from "dialect", six to seven thousand.
  - Ethnologue currently lists ***7,099*** *living* human languages.

# Human Language Speakers …

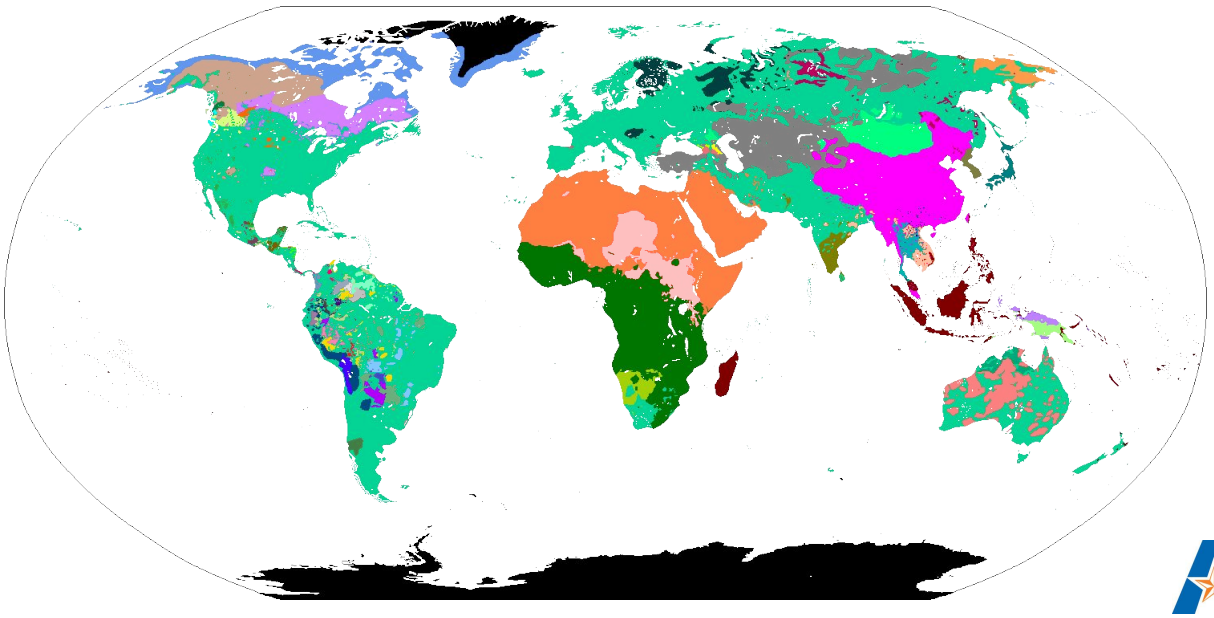
- There are ***397*** languages with at least one million speakers.
  - That's about 94% of the population and 6% of the languages.
- The rest of the languages, 94%, are spoken by only 6% of the population.

Ethnologue

| *Language* | *Native Speakers* |
|---|---|
| Mandarin | 898 M |
| Spanish | 437 M |
| English | 372 M |
| Arabic | 295 M |
| Hindi | 260 M |
| Bengali | 242 M |
| Portuguese | 219 M |
| Russian | 154 M |
| Japanese | 128 M |
| Punjabi | 93 M |

# Human Language Families

"PiMaster3" (WikiMedia Commons)
https://commons.wikimedia.org/wiki/File:Primary_Human_Language_Families_Map.png



## Why So Many Programming Languages?

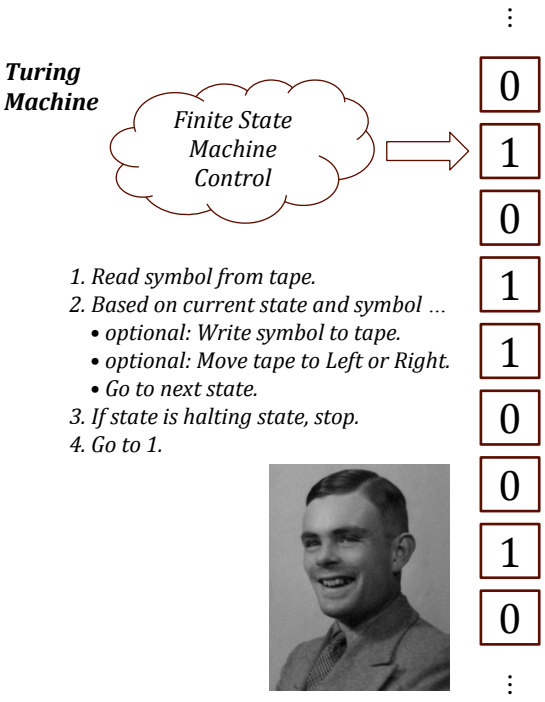
- *Evolution*: Computer Science is still a young discipline and new and better ways are still being discovered / invented.
- *Special Purposes*: Someone has a new / unique need and a new language is created to service it.
- *Personal Preference*: Different persons prefer different ways. Much of this is simply a matter of taste or historical accident.

## Why is a Programming Language Successful?

- *Expressive Power*: Formally, all the same: *Turing Complete*, but some are just plain *easier* to use in specific cases because of their features.
- *Easy to Learn*: Easier to learn, more likely to be used.
  - BASIC, Pascal, Java (kind of, when compared to C++)
- *Easy to Implement*: Can't use a language if there's no implementation. Conversely, if cheap and easy to implement, more likely to catch on.
  - BASIC, FORTH, Pascal

# [ *Turing Complete* ]

- ***Informally***, any real-world general-purpose computer or computer language can approximately simulate the *computational* aspects of any other real-word general-purpose computer or computer language.
- In other words, they are all the same; it's just that some may be more or less *convenient* for any particular *computation*.




*1. Read symbol from tape.*
*2. Based on current state and symbol …*
- *optional: Write symbol to tape.*
- *optional: Move tape to Left or Right.*
- *Go to next state.*

*3. If state is halting state, stop.*
*4. Go to 1.*



http://www.turingarchive.org/browse.php/K/7/9-16

# Why Successful?

- *Standardized*: Either formally or through a canonical implementation. Ensures portability of code across platforms.
  - Pascal hurt here because it lacked standardized support.
- *Open Source*: Free availability of language definition and compilers.
  - Free *GNU Compiler Collection* includes C, C++, Objective-C, Fortran, Ada, and Go.

## Why Successful?

- *Good Compiler*: The better the compiler, the better the apps run, the more likely the language will be used.
  - Fortran has benefited greatly from excellent compilers.
- *Economics / Patronage*: If one is *forced* to use a language, that tends to make it "successful". Also, if there's lots of money to be made writing in a particular language …
  - COBOL, Ada (DoD)
  - PL/I (IBM); C# (Microsoft); Objective-C, Swift (Apple)

## Why Successful?

- *Inertia*: Some languages have so much "legacy" code written in them, they will never go away. It's too expensive to replace them.
  - Much of the world's financial infrastructure is *still* in COBOL.
- *Good "Ecosystem"*: Not only the language and its implementation, but also the libraries, etc., around it.
  - Python

## Summary …

- Generally, no single factor for there being so many languages and for why some languages are so successful.
- *"It is what it is …"*
- *De gustibus non est disputandum.*
  (roughly, *Concerning taste, there is no argument*.)

## Points of View

- Language *Designer / Implementor*
  - Originally: The language is a *way of telling the hardware what to do*, a way of controlling the machine but more abstractly.
  - Now: The language is a *way of providing users with expressive power*. (While still telling the hardware what to do.)
- Language *User*
  - The language is a *way of thinking*, a way of expressing an algorithm, whether to another person or to a machine.

## Points of View

- Often the Designer and the User are in sync.
  - For example, both want the best possible execution speed.
- However, there are *always* conflicts and trade-offs.
  - Language features come with costs (conceptual, implementation, at run-time).
  - Suppose a feature's cost affects even programs that do not use that feature?

## Classifying Programming Languages

- *Declarative*
  - Functional: Lisp, Scheme, ML, Haskell
  - Dataflow: Id, VHDL
  - Logic / Constraint-based: Prolog, spreadsheets, SQL
- *Imperative*
  - von Neumann: C, Ada, Fortran, …
  - Object-Oriented: Smalltalk, Eiffel, Java, …
  - Scripting: Perl, Python, PHP, …

## Language Families

- *Declarative*
  - Focus is on *what* the program is to do
  - Try to get away from "irrelevant" implementation details
  - Usually end up having to express at least some *how* just to control execution
  - Unclear how good a compiler can be in this space
    - Remember, the underlying hardware is *imperative*
  - Generally needs a way for user to specify algorithm when the compiler is unable to find one automatically

## Language Families

- *Imperative*
  - Focus is on *how* the program is to accomplish its function
  - Predominate over declarative languages primarily for performance reasons
  - Line up better with the structure of the underlying hardware
  - Perhaps more intuitive for most users (debatable!)

## Models of Computation

- *Functional*: Computation model based on the recursive definition of functions, inspired by Church's *lambda calculus*. A program is a function from *inputs* to *outputs* defined in terms of simpler functions, which are defined in terms of still simpler functions, which are …

- Example languages: Lisp, ML, Haskell

## Models of Computation

- *Dataflow*: Computation model is a flow of information (*tokens*) among (primitive) functional *nodes*. A node is *triggered* by the arrival of a token as input to generate output. The computation is inherently parallel.

- Example languages: Id, VAL, SISAL.

## Models of Computation

- *Logic / Constraint-based*: Computation model based on the predicate logic, which attempts to find values that satisfy specified relationships using goal-directed search through logical rules.
- Example languages: Prolog, SQL, XSLT, Excel.

## Models of Computation

- *von Neumann*: Computation model ultimately based on the modification of variables. Functional languages have *expressions* with *values*; von Neumann languages have *statements* that affect subsequent statements via *side effects* on memory.
- Example languages: Pretty much all "traditional" languages, including Fortran, Algol, Cobol, Basic, Pascal, Ada, C, …

## Models of Computation

- *Object Oriented*: Computation model closely related to von Neumann, but more tightly controlled. Instead of changing variables, a program is an interaction among (semi-)independent *objects*, each with its own state and subroutines to modify that state.

- Example languages: Simula, Smalltalk, C++, Java
- *Functional + Object Oriented*: CLOS, OCaml, F#

## Models of Computation

- *Scripting*: Kind of a catch-all category to describe languages that are (generally) used to "stitch together" other items.
  - Job control: JCL, csh, bash, PowerShell, AppleScript
  - Dynamic web content: JavaScript, PHP
  - Embedded app control: Tcl, Lua, Python
  - "General purpose": Perl, Python, Ruby

## Programming *with* Language

- ***Machine Code***
  - The actual ones and zeros that the processor uses to control its operation.
  - Each processor architecture has its own interpretation.
- Example is a GCD routine for the x86 architecture.

```
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
```

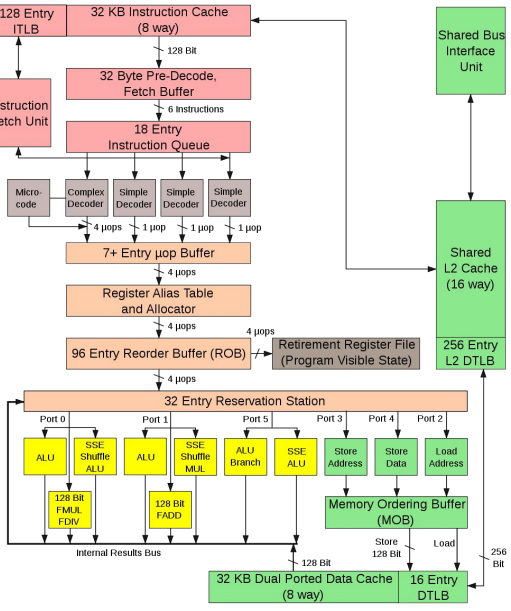
## Programming *with* Language

- ***Assembly Language***
  - Gives mnemonic (!) names to the instructions, registers, etc.
  - An *assembler* translates this into machine code.
  - Eventually included more than just 1-1 correspondence. (E.g., *macros*)
- Q: How was the first assembler written?

```
   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
```

# Machine Organization

- On the other hand, just because one *can* program in assembler, doesn't mean one *should*.
- Think you can keep everything percolating at maximum efficiency?
  - And that's a *Core 2* from 2006!




Intel Core 2 Architecture

"Appaloosa" (WikiMedia Commons)
https://commons.wikimedia.org/wiki/File:Intel_Core2_arch.svg

# Programming *with* Language

- ***High-Level Language***
  - Gets away from any particular processor architecture. (Generally …)
  - A *compiler* translates the high-level language to (assembly which is then translated to) machine code. (Generally …)
  - At first, humans could write better assembly code than the compiler generated, so slow to catch on.
  - But, reduced the number of statements that had to be written by a factor of 20!
  - It took 18 staff-years (!) to write the first FORTRAN compiler.

```
      FUNCTION IGCD()
      READ(5,500) IA, IB
500   FORMAT(2I5)
600   IF (IA.EQ.IB) GOTO 800
      IF (IA.GT.IB) GOTO 700
      IB = IB - IA
      GOTO 600
700   IA = IA - IB
      GOTO 600
800   IGCD = IA
      RETURN
      END(2, 2, 2, 2, 2)
```

- Q: How was the first compiler written?

## GCD in Different Models of Computation

- von Neumann
  - To compute the gcd of a and b, check to see if a and b are equal. If so, return one of them. Otherwise, replace the larger one by their difference and repeat.
- As an imperative model, it tells one how to compute GCD as a series of steps to be carried out.

```
int gcd(a, b)
{
  while (a != b) {
    if (a > b) {
      a = a - b;
    } else {
      b = b - a;
    }
  }

  return a;
}
```

## GCD in Different Models of Computation

- Functional
  - The gcd of `a` and `b` is defined to be
    - `a`, when `a` and `b` are equal
    - `gcd(b,a-b)`, when `a>b`
    - `gcd(a,b-a)`, when `b>a`
- GCD is defined in terms of itself.

```
let rec gcd a b =
  if a = b then a
  else if a > b then
    gcd b (a - b)
  else
    gcd a (b - a)
```

# GCD in Different Models of Computation

- Logic
  - The proposition `gcd(a, b, g)` is true if
    - `a`, `b`, and `g` are all equal
    - `a` is greater than `b` and there exists a number `c` such that `c` is `a-b` and `gcd(c, b, g)` is true
    - `a` is less than `b` and there exists a number `c` such that `c` is `b-a` and `gcd(c, a, g)` is true.
  - To compute the gcd of a given pair of numbers, search for a number `g` (and various numbers `c`) for which these rules allow one to prove that `gcd(a, b, g)` is true.

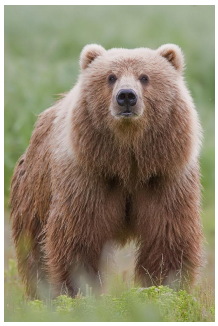
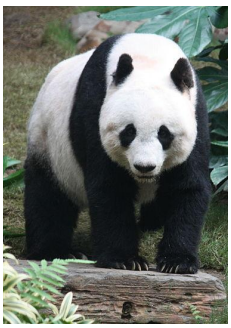
```
gcd(A,B,G) :- A = B,
              G = A.

gcd(A,B,G) :- A > B,
              C is A-B,
              gcd(C,B,G).

gcd(A,B,G) :- B > A,
              C is B-A,
              gcd(C,A,G).
```

# Observation …

- While languages can be classified into categories, families, models of computation, and so forth, the distinctions are *not* always clear-cut.
  - von Neumann, object-oriented, and scripting languages overlap greatly.
  - Functional and logic languages almost always include imperative constructs.
  - More-modern imperative languages are getting functional constructs.
  - … and so forth …

## Observation …

- *Classifying* is often quite subjective.
  - Check out the concept of *Taxonomy* in biology.
  - E.g., what's the relationship between *Bears*, *Giant Pandas*, *Red Pandas*, and *Raccoons*? Argued over for decades ...

https://commons.wikimedia.org/wiki/File:{2010-brown-bear.jpg,Grosser_Panda.JPG,RedPandaFullBody.JPG,:Procyon_lotor_(Common_raccoon).jpg}

## Observation …

- Scott's book is dominated by von Neumann and object-oriented material. That's why we have Warburton's book as well.
  - Notice Warburton discusses Java 8's Lambda Expressions, a functional feature added to an object-oriented language.
- We will spend most of our time on imperative languages …

## Observation …

- For additional examples of the same program written in different languages, check out:
  - *The Hello World Collection*
    - https://helloworldcollection.github.io/
  - *99 Bottles of Beer*
    - http://www.99-bottles-of-beer.net/

## Why Study Programming Languages?

- *Help choose an appropriate language*
  - Most languages are better for some things rather than others.
    - Fortran for scientific computation? (Or would that be C?)
    - C for systems programming? (Or would that be C++, C#, ***Lisp?!***)
    - C or C++ for embedded programming? (Or Ada?)
    - PHP or Ruby for web-based app?
    - Visual Basic or Java for GUI app? (Or Python?)
    - Lisp, Scheme, or ML for symbolic?
    - … and so on …

## [ *Lisp Machines …* ]

- Pioneered garbage collection, laser printing, windowing systems, mouse, high-resolution displays, …
- OS, tools, environment, etc. all written in Lisp!
- LM-2 (1981) was > 500 kloc.

http://www.ifis.uni-luebeck.de/~moeller/symbolics-info/symbolics-images/Symbolics-2-3.JPG

## Why Study?

- *Make it easier to learn a new language*
  - Languages are clearly grouped into families and similarities abound.
    - If you know C++, Java and C# are easier to pick up.
    - If you know Lisp, Scheme and ML aren't hard.
    - If you know ML, Haskell is within reach. Etc., etc., etc.
  - Basic concepts underlie *all* languages.
    - Most of Scott's book is about these concepts
      - Types, control, abstraction, naming, …

## Why Study?

- *Make better use of a language you already "know"*
  - Understand "obscure" or often-misunderstood features
    - In C, unions, arrays and pointers, separate compilation, varargs, catch and throw, …
    - In Lisp, first-class functions, closures, streams, catch and throw, symbol internals, …
  - Not necessarily *use-just-to-use*, but also to understand someone else's program that *does* use these features

## Why Study?

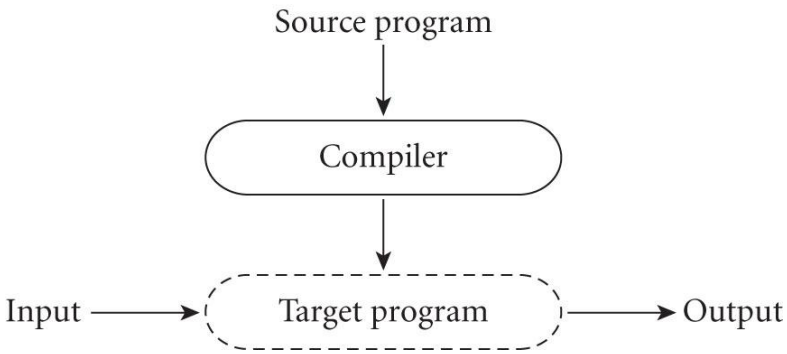
- *Make better use of a language you already "know"*
  - Understand implementation costs so you can choose wisely between alternative ways of doing things.
  - Understand "quirks" of the language.
  - Base decisions on how feature will be implemented.
  - This can be controversial in some cases …
    - Code should be clear, not "clever" to decrease error rate and improve maintainability
    - Implementations can and do change

## Why Study?

- *Make better use of a language you already "know"*
  - How to "fake" a feature, for example:
    - Older versions of Fortran lacked good control structures, recursion, etc. Programmer *self-discipline* (we call it *patterns* now) had to suffice.
    - Naming conventions, regularized commenting can be used to imitate modules in languages with poor abstraction capabilities.
    - Structured use of subroutines and static variables can be used to imitate the iterators of Clu, Python, C#, Ruby, etc.

## Compiling a Program

- A *compiler* translates a *source* program into an equivalent *target* program and then goes away. At some later time, the target program can be executed.

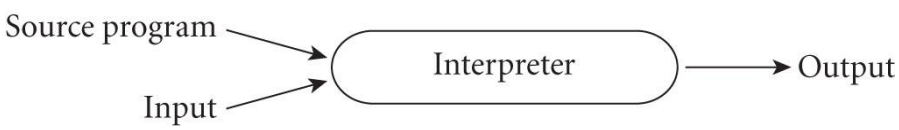



## Compiling a Program

- During compilation, the compiler is the *locus of control*.
- During its own execution, the target program is the locus of control.
- Typically, the compiler is a machine language program.
- Typically, the target program is also a machine language program.
- Generally leads to best performance.
- *Translate once, run many times*.

## Interpreting a Program

- An *interpreter*, on the other hand, typically stays around for the execution of the target program.
  - The interpreter is the locus of control during all parts of the execution.
  - The interpreter is in effect a *virtual machine* whose machine language is the programming language.
  - *Translate every time*.




## Interpreting a Program

- Generally, interpretation is more flexible and has better diagnostics than compilation.
  - The source code of the program is still available.
- Some language features are very difficult to implement without interpretation.
  - "On-the-fly" code generation

## [ *Read-Eval-Print Loop (REPL)* ]

- A *Read-Eval-Print Loop* is an interactive environment wherein the user types a line at a time which is then evaluated and results displayed.
  - *Read* — Get input from user.
  - *Eval* — Determine value.
  - *Print* — Display result to user.
  - *Loop* — Do again, until terminated.
- Can be created for any text-based language, Lisp, Python, Java …
- REPL is a one-liner in Lisp for Lisp:

  `(loop (print (eval (read))))`
- Lot of REPLs available *live* and *on-line*:

  `http://joel.franusic.com/Online-REPs-and-REPLs/`

```
dalioba@svr-test-01:~$ python
Python 3.7.1 (default, Dec 14 2018, 19:28:38)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 1+2
3
>>> a = 10
>>> b = 5
>>> a+b
15
>>> c = a + b
>>> c
15
>>> b = "Hi, there!"
>>> c = a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> for i in range( 5 ) :
...     print( i, i*i )
...
0 0
1 1
2 4
3 9
4 16
>>> quit()
dalioba@svr-test-01:~$
```

## Compilation vs. Interpretation

- In both cases, instructions are executed on the target processor.
- A compiler generates instructions by analyzing the source code in its *entirety* and optimizes the generated code based on the *entire source code* and specific optimizations.
- The generation of the instructions is (generally) independent of the execution of the target program.

## Compilation vs. Interpretation

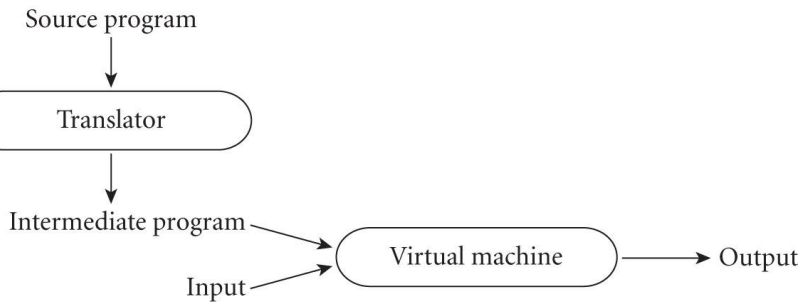
- An interpreter typically reads one “line” at a time.
  - It cannot perform overall analysis of the code as it does not know all of the code.
  - It therefore executes one line at a time without optimization.
- It must read the source code and generate instructions as it is executing the target program.
- It may take several operations for an interpreter to accomplish the same operation a compiler would have generated one machine instruction for.

## Compilation vs. Interpretation

- Interpretation
  - Don't have to wait for compile and link steps.
  - Usually takes less space than a compiler.
  - Much easier to port an interpreter to a new architecture.
- Compilation
  - Generally much better performance.
  - Generally catches many errors earlier, before the target program even executes.

## Compiling *and* Interpreting a Program

- While compilation and interpretation are different, they can be used together.




## Compiling *and* Interpreting a Program

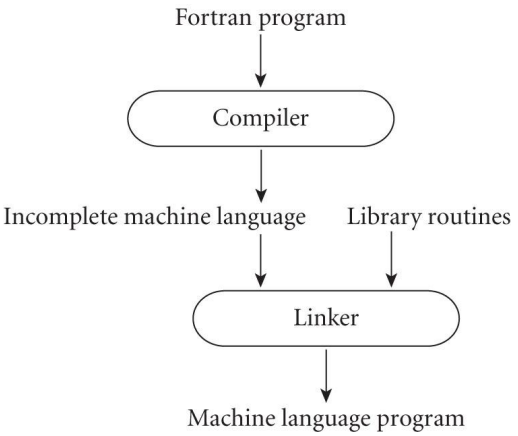
- If the initial translator is "simple", we'd still call this interpretation. If it's "complex" we'd call this compilation.
  - Subjective!
  - Also, both parts can be quite complex, as in the case of Java.
- Instead, we'll use "compiling" to mean that a *complete analysis* of the source code is done by the translator.
  - Not just a "mechanical" transformation, as, e.g., cpp does.
- Also, the intermediate program would not bear a strong resemblance to the source code. It's a *nontrivial* translation.

## Pure Interpretation

- The earliest implementations of Basic were close to pure interpreters.
  - The original characters were read and reread and reread as the source code was interpreted. Removing comments sped up the program's execution!
- Generally, a modern interpreter processes the source code to remove whitespace and comments, characters are grouped into *tokens*, and even perhaps identify syntactic structures.

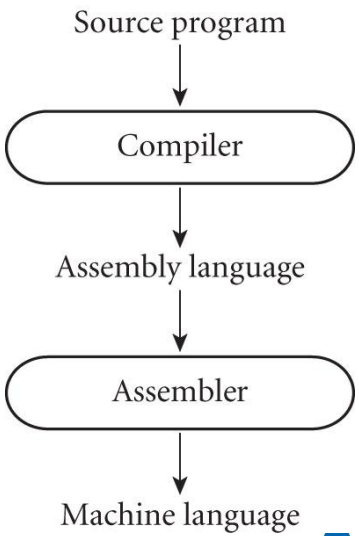
## Pure Compilation

- (Early) Fortran implementations however were close to pure compilation.
  - The source code is translated into machine code.
  - May have a *library* of subroutines for shared use.
  - A *linker* puts it all together.
- There's still *some* interpretation
  - **FORMAT** statements



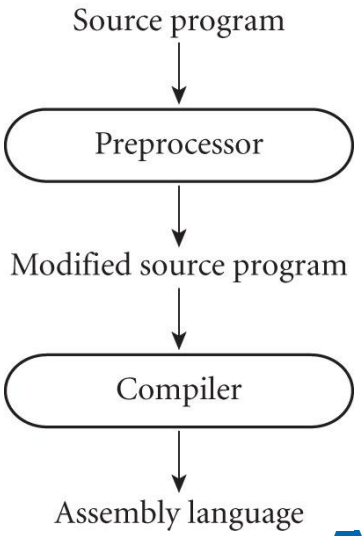

## Chain of Compilation

- Many compilers generate assembly language instead of machine code.
  - Can make debugging easier
  - Helps isolate compiler from changes in the operating system



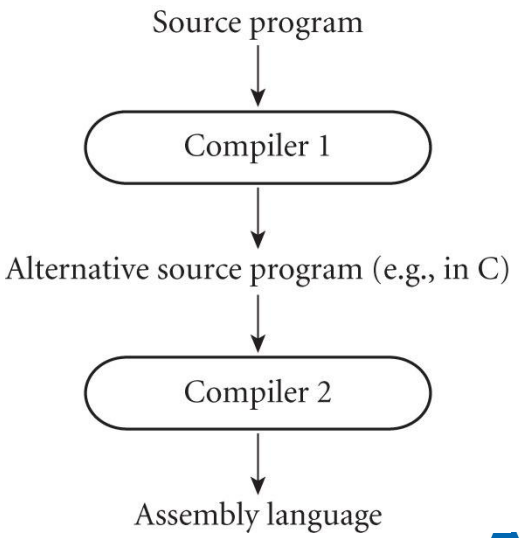

## Chain of Compilation

- Many compilers (e.g., C) begin with a *textual* preprocessor that
  - Removes comments
  - *Macro* expansion
  - Provides *insertion* (`#include`) and *conditional compilation* (`#if`) mechanisms




## Chain of Compilation

- Some compilers generate a relatively high-level intermediate program and then use *another* compiler to get to the final target program.
  - Called *source-to-source* compiling, C++ was originally implemented this way.
  - Still considered a true compiler!
    - Full analysis, non-trivial transform




## Self-Hosting Compiler

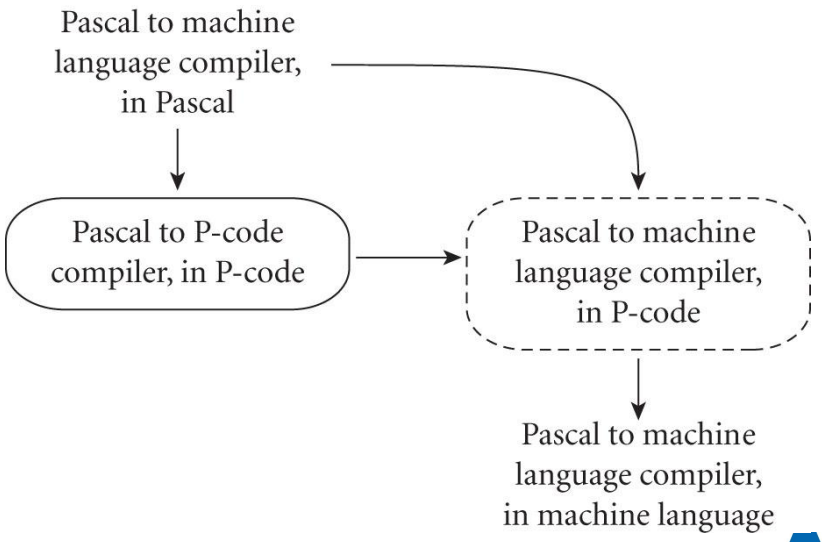
- A *self-hosting* compiler is written in its own language.
  - An Ada compiler written in Ada, a C compiler written in C, and so forth.
- So how to compile it the first time?
  - *Bootstrapping*: Start with a very simple implementation that knows just enough to get to the next level.
    - Often the earliest parts are *interpreters*.
  - Each step up adds more capability until the entire compiler can be compiled.

## Self-Hosting Compiler Example

The original Pascal distribution included:

- A Pascal compiler, written in Pascal, that generates *P-code*.
- The same compiler, already in P-code.
- A P-code interpreter written in Pascal.

So, translate the P-code interpreter into a local language, then use it to run the P-code version of the compiler, then compile the Pascal version of the compiler.




## [ *P-Code* ]

- A “P-Code” is a very simple language that is easy to translate to machine code.
  - “P” for *portable* or *pseudo*.
  - Used to make it easier to port software from one machine to another.
  - Also used to isolate compiler front ends from back ends.
- P-Code can be considered an intermediate form in the compilation process.

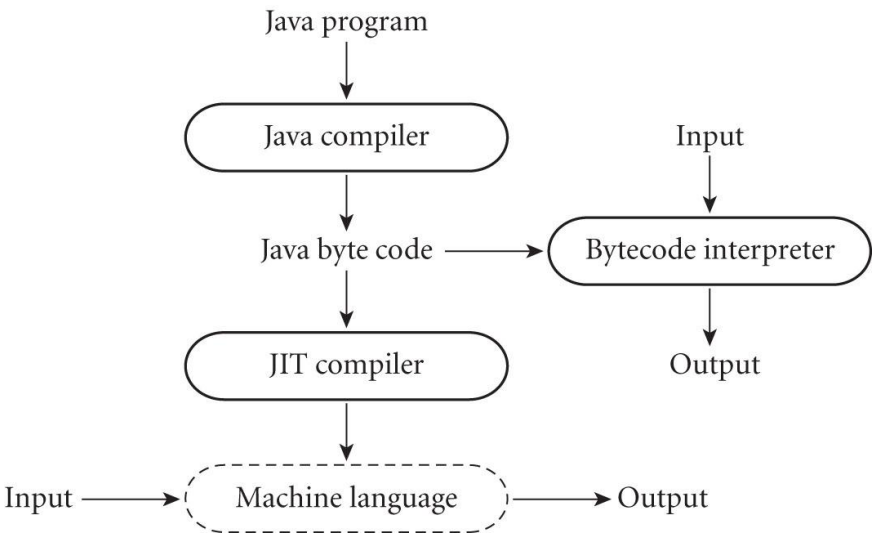
## Compiling Interpreted Languages

- Some programs in traditionally interpreted languages can be compiled under a set of assumptions (about, e.g., types, bindings)
  - If at run-time the assumptions are violated, the program drops back into interpreted mode.
  - Otherwise, the performance advantages of compiled code can be obtained.

## Just-In-Time Compilation

- An extension of this concept is *JIT*.
- Compilation can be *delayed* until the last possible moment.
- Allows for handling "on-the-fly" code, optimization based on run-time conditions, etc.
- Also supports platform-independent intermediate representation that can be distributed and then compiled locally into machine code.
- Examples include Java, C#.

## Just-In-Time Compilation




## Programming Environments

- Compilers and interpreters do not exist in isolation.
- Many other tools fit together to support the programming process.

| 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 | `ld, ld-so` |
| Perusal tools | `More, less, od, nm` |
| Program cross-reference | `ctags` |

## Integrated Development Environments (IDEs)

- Command-line tools are great as far as they go, but programmer effectiveness, productivity, and efficiency can be increased by using an Integrated Development Environment (IDE).
  - Better connectivity among the tools, dynamic and responsive environment, built-in language knowledge and support, etc. Highly interactive.
- Not a new idea!
  - Smalltalk and Lisp, among others, had IDEs by the 1980s.

## Overview of Compilation

- Compilation is one of the most well-studied aspects of computer science.
- Much of compilation has been studied so much that what used to be very hard is now fairly straightforward (not to say *easy*).
  - The first Fortran compiler cost 18 staff-years.
  - Now, writing a "compiler" is a semester project for an undergraduate.

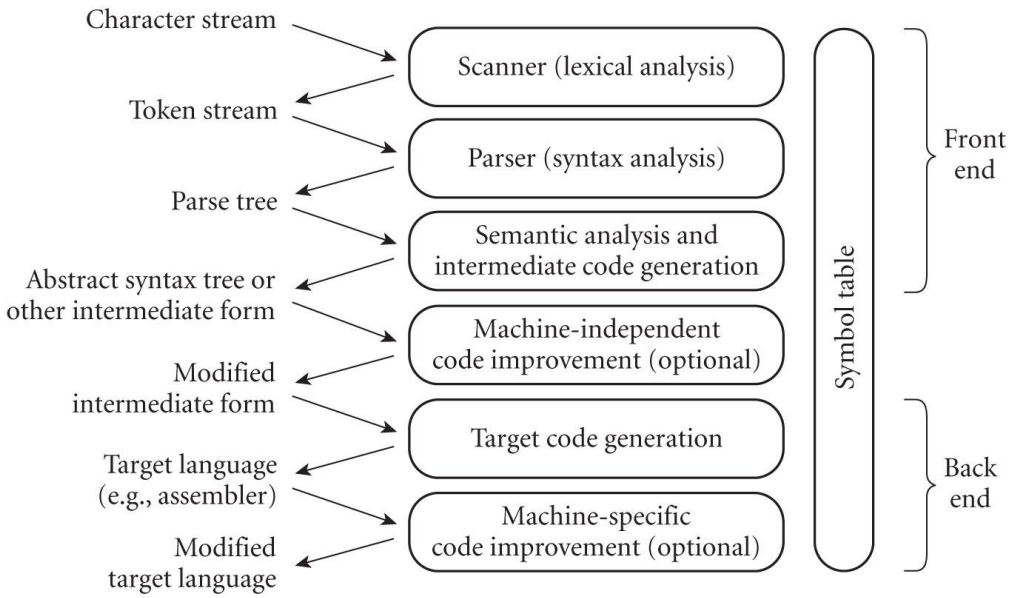
## Overview of Compilation

- This is *not* a compiler class, though, so we will consider the internals of a compiler at a high level and somewhat simply.
- The compilation process is divided into *phases*
  - Each phase determines new information to be used later or transforms what has already been learned for later use.
  - The first phases analyze the source program to discover its meaning. This is the compiler *front end*.
  - The last phases construct (and possibly improve) the target program. This is the compiler *back end*.

## Overview of Compilation

- One may hear of compiler *passes*
  - A *pass* is a phase or set of phases that are serialized with the respect to the others.
  - It does not start until all previous phases / passes have completed and it runs to completion before subsequent phases / passes start.
  - Sometimes a pass may be an entirely separate program, reading a description from and writing one to files.
  - Passes can be used to share parts of the compilation process.

## Phases of Compilation



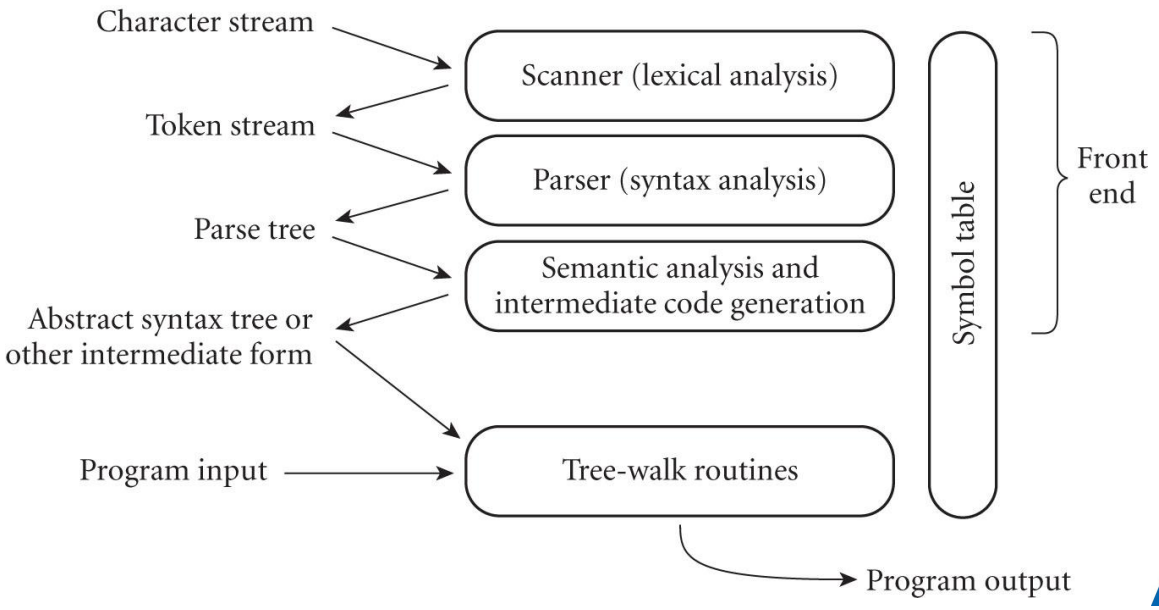

## Front End vs Back End

- The *front end* of a compiler (or interpreter) comprises the lexical, syntactic and semantic analysis steps along with the generation of intermediate code.
- The *back end* of a compiler comprises the optimization of the intermediate code, target code generation, and machine-specific code optimization steps.
  - Interpreters don't have back ends.
    - (At least not the way compilers do.)
- The front end is concerned with determining what the program means. The back end with producing the corresponding target code representation.

## Overview of Interpretation

- An interpreter shares the general structure of a compiler up through the front end.
- However, instead of generating a target program, the interpreter "executes" the intermediate form directly.
  - This execution is normally accomplished by a set of mutually-recursive routines that traverse ("walk") the abstract syntax tree, executing its nodes in order.
  - Another technique is to generate *bytecode* and hand it off to a bytecode engine. (This is getting closer to compilation.)

# Phases of Interpretation




# Overview of Compilation

- Lexical Analysis (*Scanning*)
  - The *scanner* (or *lexer*) reads individual characters and groups them into *tokens*.
  - The tokens are the smallest meaningful units of a program.

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

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

## Overview of Compilation

- Syntactic Analysis (*Parsing*)
  - Organizes the tokens into a *parse tree* that hierarchically represents higher-level constructs in terms of their lower-level parts.
  - Each construct is a *node*, its parts are its *children*, the *leaves* are the tokens from the lexical analysis phase.
  - The tree is constructed from the tokens by means of a set of recursive rules known as a *context-free grammar*.
  - The grammar defines the *syntax* of the language.

## Overview of Compilation

- Context-free grammar excerpt, from C.
  - Shows the syntax of the `while` statement.
  - ε represents the empty string.
- Quite complicated!
- In general, parse trees have an immense amount of information, even for relatively small programs.

*iteration-statement* ⟶ `while` ( *expression* ) *statement*

*statement* ⟶ *compound-statement*
*compound-statement* ⟶ { *block-item-list_opt* }

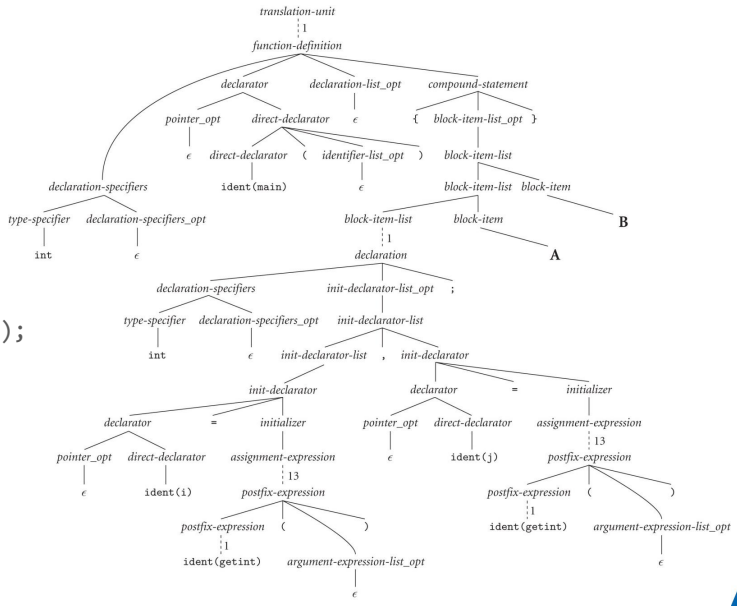
*block-item-list_opt* ⟶ *block-item-list*
*block-item-list_opt* ⟶ ε

*block-item-list* ⟶ *block-item*
*block-item-list* ⟶ *block-item-list block-item*
*block-item* ⟶ *declaration*
*block-item* ⟶ *statement*

# Overview of Compilation

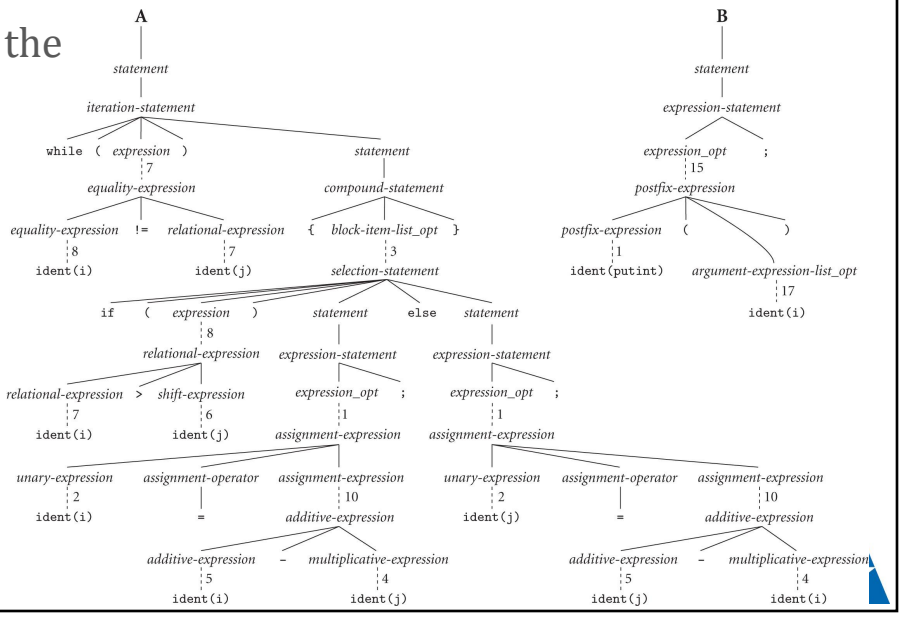
- Parse Tree for the GCD example (Part 1)

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




# Overview of Compilation

- Parse Tree for the GCD example (Part 2)



## Overview of Compilation

- Semantic Analysis
  - Discovers the *meaning* of the source program. For example,
    - Determines when the use of the same identifier means the same program entity.
    - Ensures uses are consistent, both as a test of the legality of the source program and to guide generation of code in the back end.
  - Builds and maintains a *symbol table* to map each identifier to the information known about it, including
    - Type, internal structure (if any), scope, ...

## Overview of Compilation

- Semantic Analysis
  - Enforces a large variety of rules that are not captured by the syntactic structure of the program, for example (in C),
    - Declaration of identifier before use
    - No use of identifier in an inappropriate context
    - Correct number / type of parameters in subroutine calls
    - Distinct, constant labels on the branches of a switch statement
    - Non-`void` return type function returns a value explicitly
  - These example rules are *static semantic* checks as they depend on only the structure of the program as it is written and can be enforced at compile time.

## Overview of Compilation

- Semantic rules that can't be enforced until runtime are *dynamic semantics*, for example,
  - Variables are not used in an expression unless they have been assigned a value.
  - Pointers are not dereferenced unless they refer to a valid address.
  - Array subscripts are within bounds.
  - Arithmetic expressions do not overflow.

## Overview of Compilation

- Those dynamic semantics rules just given do *not* apply to C.
  - It would have saved a lot of trouble and debugging time if they were automatically enforced.
  - Q: Why aren't they?
- C has very little in the way of dynamic semantics rule enforcement.

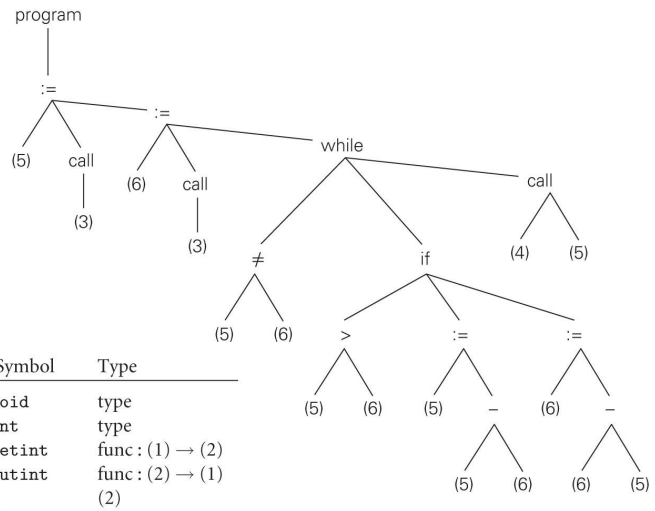
# Overview of Compilation

- The parse tree is sometimes known as a *concrete syntax tree* because it completely shows how a sequence of tokens was derived using the CFG (Context-Free Grammar).
- As the static semantics phase runs, it transforms the parse tree to an *abstract syntax tree (AST)* which retains only the essential information.
- Annotations are also made with information useful to the rest of the process.
- Nodes get *attributes* added to them as required.

# Overview of Compilation

- Abstract Syntax Tree and symbol table for GCD program

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




| 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) |

## Overview of Compilation

- It's common for an interpreter to use the AST as its representation of the program to run.
- "Execution" then amounts to a traversal of the AST by a *tree-walker* routine that takes the appropriate action at each node.

## Overview of Compilation

- Many compilers use the AST as the *intermediate form (IF)* handed off to the back end for code generation.
- Others compilers *tree walk* the AST and generate a different intermediate form.
  - A common IF is a *control-flow graph*, whose nodes are fragments of assembly language for an idealized machine.

# Overview of Compilation

- Target Code Generation
  - Translates the intermediate form into the target language.
  - Generating code that *works* is not all that hard.
  - Generating *good* code is trickier.

```
    pushl   %ebp            # \
    movl    %esp, %ebp      #  ) reserve space for local variables
    subl    $16, %esp       # /
    call    getint          # read
    movl    %eax, -8(%ebp)  # store i
    call    getint          # read
    movl    %eax, -12(%ebp) # store j
A:  movl    -8(%ebp), %edi  # load i
    movl    -12(%ebp), %ebx # load j
    cmpl    %ebx, %edi      # compare
    je      D               # jump if i == j
    movl    -8(%ebp), %edi  # load i
    movl    -12(%ebp), %ebx # load j
    cmpl    %ebx, %edi      # compare
    jle     B               # jump if i < j
    movl    -8(%ebp), %edi  # load i
    movl    -12(%ebp), %ebx # load j
    subl    %ebx, %edi      # i = i - j
    movl    %edi, -8(%ebp)  # store i
    jmp     C
B:  movl    -12(%ebp), %edi # load j
    movl    -8(%ebp), %ebx  # load i
    subl    %ebx, %edi      # j = j - i
    movl    %edi, -12(%ebp) # store j
C:  jmp     A
D:  movl    -8(%ebp), %ebx  # load i
    push    %ebx            # push i (pass to putint)
    call    putint          # write
    addl    $4, %esp        # pop i
    leave                   # deallocate space for local variables
    mov     $0, %eax        # exit status for program
    ret                     # return to operating system
```

# Overview of Compilation

- Code Improvement
  - Often referred to as *optimization*, though that's a bit presumptuous.
  - Not required, but often done in an attempt to improve the generated code.
  - In this case, the optimizer did fairly well.
    - Got rid of most loads and stores as it was able to keep the values in the registers.

```
    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
```

## Overview of Compilation

- Code Improvement
  - The previous optimization was at the target language level.
  - Another place code improvement can happen is much earlier in the compilation process, right after semantic analysis.
  - Generally, the earlier an optimization can be made, the greater the effect (improvement, we hope) on the final target program.

## Review Questions (1)

1. What is the difference between *machine language* and *assembly language*?

2. In what way(s) are high-level languages an improvement on assembly language? In what circumstances does it still make sense to program in assembler?

3. Why are there so many programming languages?

4. What makes a programming language successful?

5. Name three languages in each of the following categories: *von Neumann*, *functional*, *object-oriented*. Name two *logic* languages. Name two widely-used *concurrent* languages.

## Review Questions (2)

6. What distinguishes *declarative* languages from *imperative* languages?
7. What organization spearheaded the development of *Ada*?
8. What is generally considered the first high-level programming language?
9. What was the first functional programming language?
10. Why aren't concurrent languages listed as a separate family?

## Review Questions (3)

11. Explain the distinction between *interpretation* and *compilation*. What are the comparative advantages and disadvantages of the two approaches?
12. Is Java compiled or interpreted (or both)? How do you know?
13. What is the difference between a *compiler* and a *preprocessor*?
14. What was the intermediate form employed by the original AT&T C++ compiler?
15. What is *P-code*?
16. What is *bootstrapping*?
17. What is a *just-in-time* compiler?

## Review Questions (4)

18. Name two languages in which a program can write new pieces of itself "*on-the-fly*".

19. Briefly describe three "*unconventional*" compilers—compilers whose purpose is *not* to prepare a high-level program for execution on a processor.

20. Describe six kinds of tools that commonly support the work of a compiler within a larger programming environment.

21. Explain how an *integrated development environment* differs from a collection of command-line tools.

## Review Questions (5)

22. List the principal *phases* of compilation and describe the work performed by each.

23. List the phases that are also executed as part of interpretation.

24. Describe the form in which a program is passed from the *scanner* to the *parser*; from the *parser* to the *semantic analyzer*; from the *semantic analyzer* to the *intermediate code generator*.

25. What distinguishes the *front end* of a compiler from the *back end*?

26. What is the difference between a *phase* and a *pass* of compilation? Under what circumstances does it make sense for a compiler to have *multiple* passes?

## Review Questions (6)

27. What is the purpose of the compiler's *symbol table*?

28. What is the difference between *static* and *dynamic* semantics?

29. On modern machines, do assembly language programmers still tend to write better code than a good compiler can? Why or why not?