LECTURE 1

Overview and History

COURSE OBJECTIVE

Our ultimate objective in this course is to provide you with the knowledge and skills necessary to create a new programming language (at least theoretically).

Let’s say I asked you to do that right now. You might ask yourself:

- How can I express the rules of my language?
- Should it be object-oriented, procedural, or are there better options?
- How do I create a compiler for the language? Should it even be a compiled language?
- Semicolons or not?

COURSE TOPICS

Language concepts

- Classification of programming languages.
- **Common language constructs:** sequencing, loops, conditions, etc.
- **Names, Scopes, and Bindings:** How and when bindings for local names are defined in languages with scoping rules.
- **Control Flow:** How programming constructs define control flow and how the choice of constructs can affect programming style.
- **Subroutines and Parameter Passing:** How the subroutine calling mechanism is implemented and how and when parameters are passed and evaluated.
- **Exception Handling:** How to improve the robustness of programs.

COURSE TOPICS

Language implementations

- Common techniques used in compilers and interpreters.
- **Lexical analysis**: Identifying correct words in a program.
- **Syntax analysis:** Identifying syntactically correct program structures.
- **Semantics analysis:** Identifying meaningful programs.

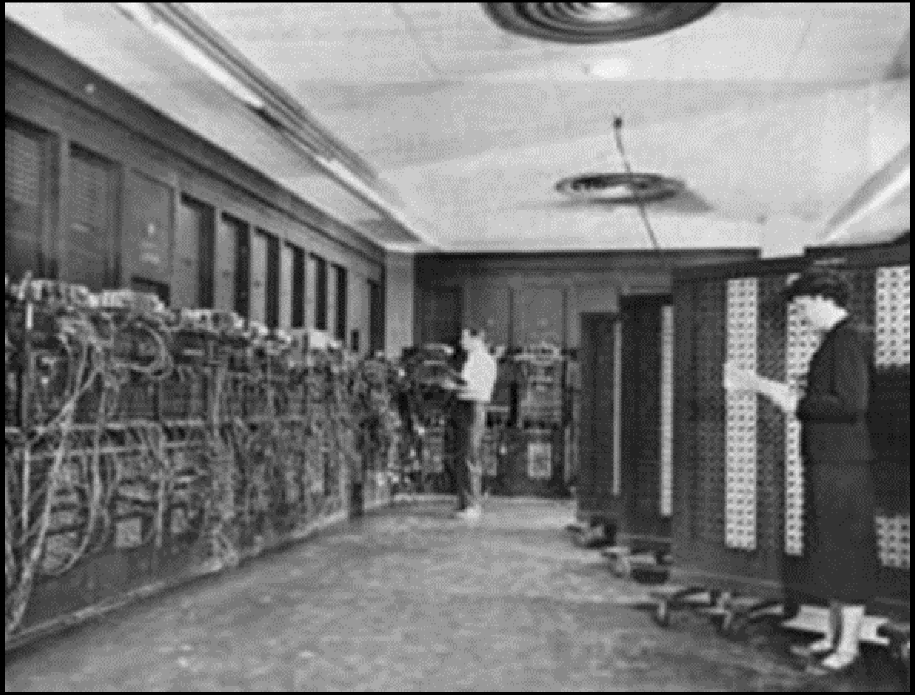
Alternative programming

- **Functional Programming**: Programming with Scheme
- **Logic Programming**: Programming with Prolog

HISTORY OF PROGRAMMING LANGUAGES

1940s: The first electronic computers were monstrous contraptions

- Programmed in binary machine code by hand via switches and later by card readers and paper tape readers.
- Code is not reusable or relocatable.
- Computation and machine maintenance were difficult: machines had short mean-time to failure (MTTF) because vacuum tubes regularly burned out.
- The term "bug" originated from a bug that reportedly roamed around in a machine causing short circuits.



MACHINE LANGUAGE PROGRAMS

Many early machines required programs to be expressed in machine language.

```
55 89 e5 53
83 ec 04 83
```

← Could you tell me what's going on here?

Programs written for the ENIAC (1946) were written in the language of the machine itself.

- A program for adding two numbers essentially outlined instructions for configuring the ENIAC's many large plugboards and switchboards.

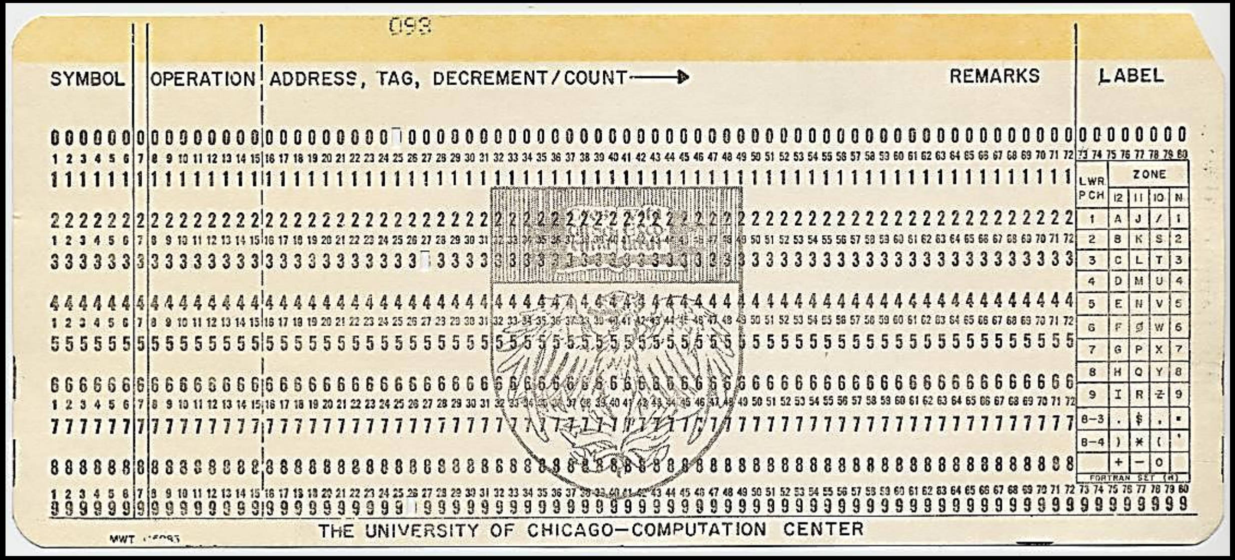
This is tedious and difficult to create and maintain.

ASSEMBLY LANGUAGE PROGRAMS

Assembly Language Programs were developed so that machine operations could be expressed in mnemonic abbreviations.

- Enables larger, reusable, and relocatable programs.
- Machine code is produced by assembler, not by programmer.
- Translation from assembly to machine is pretty much one-to-one.

Here is an example punched card for the IBM 709x Assembly Language.


SYMBOL
OPERATION
ADDRESS, TAG, DECREMENT/COUNT
REMARKS
LABEL
ZONE
THE UNIVERSITY OF CHICAGO—COMPUTATION CENTER

ASSEMBLY LANGUAGE EXAMPLE

You've all seen MIPS before.

Here's an example MIPS assembly program to compute GCD.

Easier to understand, but it might take you quite a bit of time to tell me what's going on here.

```
    addiu   sp,sp,-32
    sw      ra,20(sp)
    jal     getint
    nop
    jal     getint
    sw      v0,28(sp)
    lw      a0,28(sp)
    move    v1,v0
    beq     a0,v0,D
    slt     at,v1,a0
A:  beq     at,zero,B
    nop
    b       C
    subu    a0,a0,v1
B:  subu    v1,v1,a0
C:  bne     a0,v1,A
    slt     at,v1,a0
D:  jal     putint
    nop
    lw      ra,20(sp)
    addiu   sp,sp,32
    jr      ra
    move    v0,zero
```

ASSEMBLY LANGUAGE EXAMPLE

Example MIPS R4000 machine code of the assembly program:

```
27bdffd0 afbf0014 0c1002a8 00000000 0c1002a8 afa2001c 8fa4001c
00401825 10820008 0064082a 10200003 00000000 10000002 00832023
00641823 1483fffa 0064082a 0c1002b2 00000000 8fbf0014 27bd0020
03e00008 00001025
```

HIGH-LEVEL PROGRAMMING LANGUAGES

Mid 1950's: Development of FORTRAN (FORmula TRANslator), the first higher-level programming language.

- Mainly developed for solving numerical problems.

Other high-level languages soon followed.

- Algol-58, COBOL, Lisp, BASIC, C

Important result: programming is now a machine-independent task.

- High-level Source Code → Intermediate Representation → Machine Code

FORTRAN 77

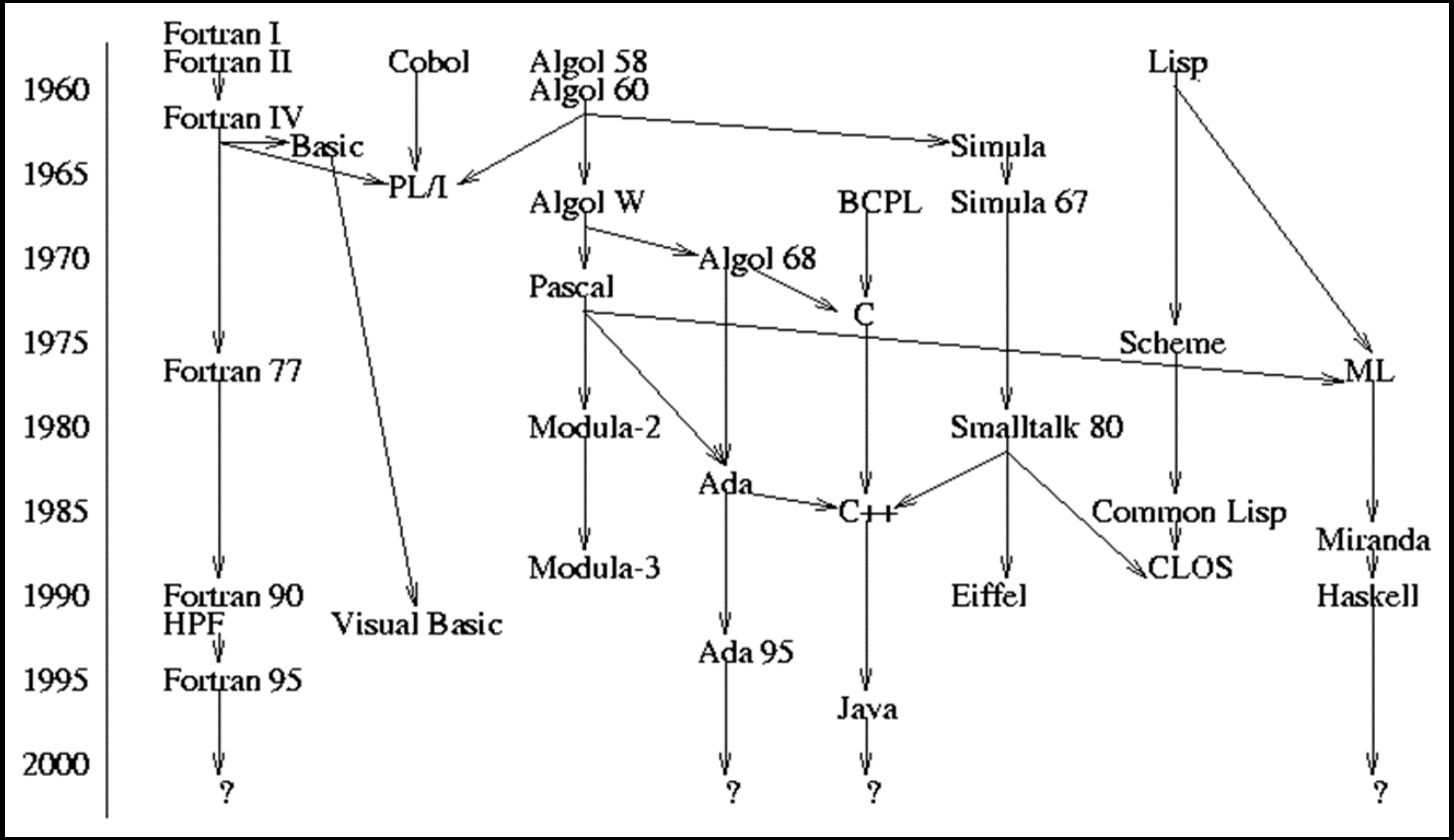
FORTRAN is still widely used for scientific, engineering, and numerical problems.

FORTRAN 77 includes:

- Subroutines, if-then-else, do-loops.
- Types (primitive and arrays).
- Variable names are upper case and limited to 6 chars.
- No recursion.
- No structs/classes, unions.
- No dynamic allocation.
- No case-statements and no while-loops.

```
      PROGRAM GCD
C     variable names that start with
C           I,J,K,L,N,M are integers
C     read the parameters
      READ (*, *) I, J
C     loop while I!=J
10    IF I .NE. J THEN
      IF I .GT. J THEN
      I = I - J
      ELSE
      J = J - I
      ENDIF
      GOTO 10
      ENDIF
C     write result
      WRITE (*, *) ’GCD =’, I
      END
```

GENEALOGY OF PROGRAMMING LANGUAGES



PROGRAMMING LANGUAGES

Until now, you’ve likely only seen C++, C, and perhaps Java. Some of you may have experience with Python, PHP, JavaScript, etc.

There are A LOT of languages out there – some of them are undoubtedly completely different than anything you’ve seen before.

WHY ARE THERE SO MANY PROGRAMMING LANGUAGES?

Evolution

- Design considerations: What is a good or bad programming construct?
- Early 70s: structured programming, in which goto-based control flow was replaced by high-level constructs (e.g. while loops and case statements).
- Late 80s: nested block structure gave way to object-oriented structures.

WHY ARE THERE SO MANY PROGRAMMING LANGUAGES?

Special Purposes

- Many languages were designed for a specific problem domain, e.g:
  - Scientific applications
  - Business applications
  - Artificial intelligence
  - Systems programming
  - Internet programming

Personal Preference

- The strength and variety of personal preference makes it unlikely that anyone will ever develop a universally accepted programming language.

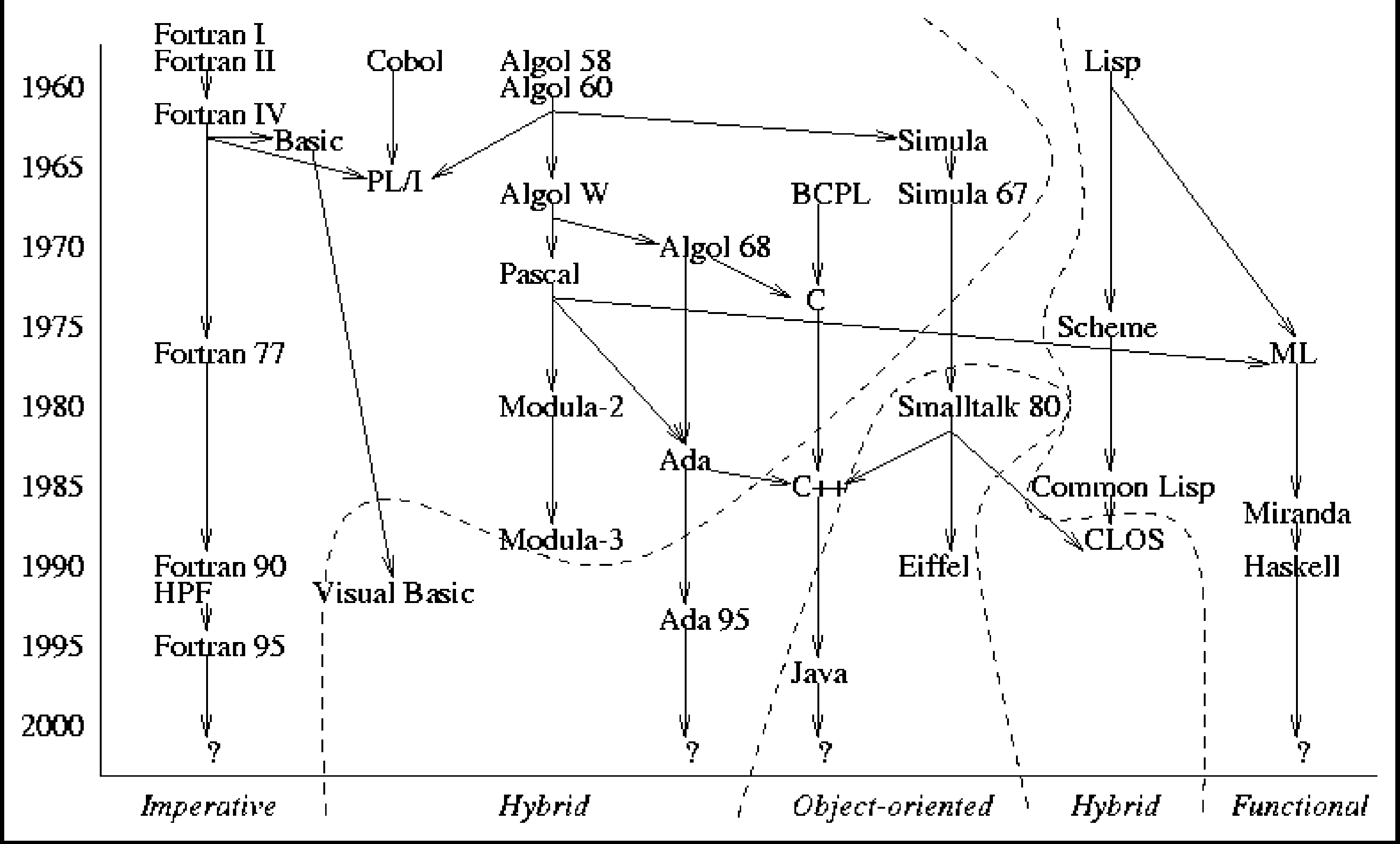
WHAT MAKES A PROGRAMMING LANGUAGE SUCCESSFUL?

- Expressive Power
  - Theoretically, all languages are equally powerful (Turing complete).
  - Language features have a huge impact on the programmer's ability to read, write, maintain, and analyze programs.
- Ease of Use for Novice
  - Low learning curve and often interpreted, e.g. Basic and Logo.
- Ease of Implementation
  - Runs on virtually everything, e.g. Basic, Pascal, and Java.

WHAT MAKES A PROGRAMMING LANGUAGE SUCCESSFUL?

- Open Source
  - Freely available, e.g. Java.
- Excellent Compilers and Tools
  - Supporting tools to help the programmer manage very large projects.
- Economics, Patronage, and Inertia
  - Powerful sponsor: Cobol, PL/I, Ada.
  - Some languages remain widely used long after "better" alternatives.

CLASSIFICATION OF PROGRAMMING LANGUAGES



CLASSIFICATION OF PROGRAMMING LANGUAGES

- Declarative: Implicit solution. What should the computer do?
  - Functional
    - Lisp, Scheme, ML, Haskell
  - Logic
    - Prolog
  - Dataflow
    - Simulink, Scala
- Imperative: Explicit solution. How should the computer do it?
  - Procedural
    - Fortran, C
  - Object-Oriented
    - Smalltalk, C++, Java

Note that these classifications aren’t entirely rigid. Languages can have multiple classifications.

CONTRASTING EXAMPLES

Procedural (C):

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

Functional (Haskell):

```
gcd a b
  | a == b = a
  | a >  b = gcd (a-b) b
  | a <  b = gcd a (b-a)
```

Logical (Prolog):

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

NEXT LECTURE

We will start our course by discussing compilation and interpretation.

- Compilation and interpretation.
- Virtual machines.
- Static linking and dynamic linking.
- Compiler in action (g++).
- Integrated development environments.