ICOM 4036

Structure and Properties of Programming Languages

Lecture 1

Prof. Bienvenido Velez
Spring 2008

Some slides adapted from Sebesta's Concepts of Programming Languages

Outline

- Motivation
- Programming Domains
- Language Evaluation Criteria
- Influences on Language Design
- Language Categories
- Language Design Trade-Offs
- Implementation Methods
- Milestones on PL Design

What is a Programming Language?

- A Programming Language …
  - ... provides an encoding for algorithms
  - …should express all possible algorithms
  - ... must be decodable by an algorithm
  - ... should support complex software
  - …should be easy to read and understand
  - ... should support efficient algorithms
  - …should support rapid software development

Motivation:
Why Study Programming Languages?

- Increased ability to express ideas
- Improved background for choosing appropriate languages
- Greater ability to learn new languages
- Understand significance of implementation
- Ability to design new languages
- Overall advancement of computing

Programming Domains

- Scientific applications
  - Large number of floating point computations
- Business applications
  - Produce reports, use decimal numbers and characters
- Artificial intelligence
  - Symbols rather than numbers manipulated. Code = Data.
- Systems programming
  - Need efficiency because of continuous use. Low-level control.
- Scripting languages
  - Put a list of commands in a file to be executed. Glue apps.
- Special-purpose languages
  - Simplest/fastest solution for a particular task.

Language Evaluation Criteria

- Readability
- Write-ability
- Reliability
- Cost
- Others

The key to good language design consists of crafting the best possible compromise among these criteria

Language Evaluation Criteria Readability

- Overall simplicity
  - Too many features is bad
  - Multiplicity of features is bad
- Orthogonality
  - Makes the language easy to learn and read
  - Meaning is context independent
  - A relatively small set of primitive constructs can be combined in a relatively small number of ways
  - Every possible combination is legal
  - Lack of orthogonality leads to exceptions to rules

Language Evaluation Criteria
Write-ability

- Simplicity and orthogonality
- Support for abstraction
- Support for alternative paradigms
- Expressiveness

Language Evaluation Criteria Reliability

Some PL features that impact reliability:

- Type checking
- Exception handling
- Aliasing

Language Evaluation Criteria
Cost

What is the cost involved in:

- Training programmers to use language
- Writing programs
- Compiling programs
- Executing programs
- Using the language implementation system
- Risk involved in using unreliable language
- Maintaining programs

Language Evaluation Criteria Other

- Portability
- Generality
- Well-definedness
- Elegance
- Availability
- …

Some Language Design Trade-Offs

- Reliability vs. cost of execution
- Readability vs. writability
- Flexibility vs. safety

Influences on Language Design Through the Years

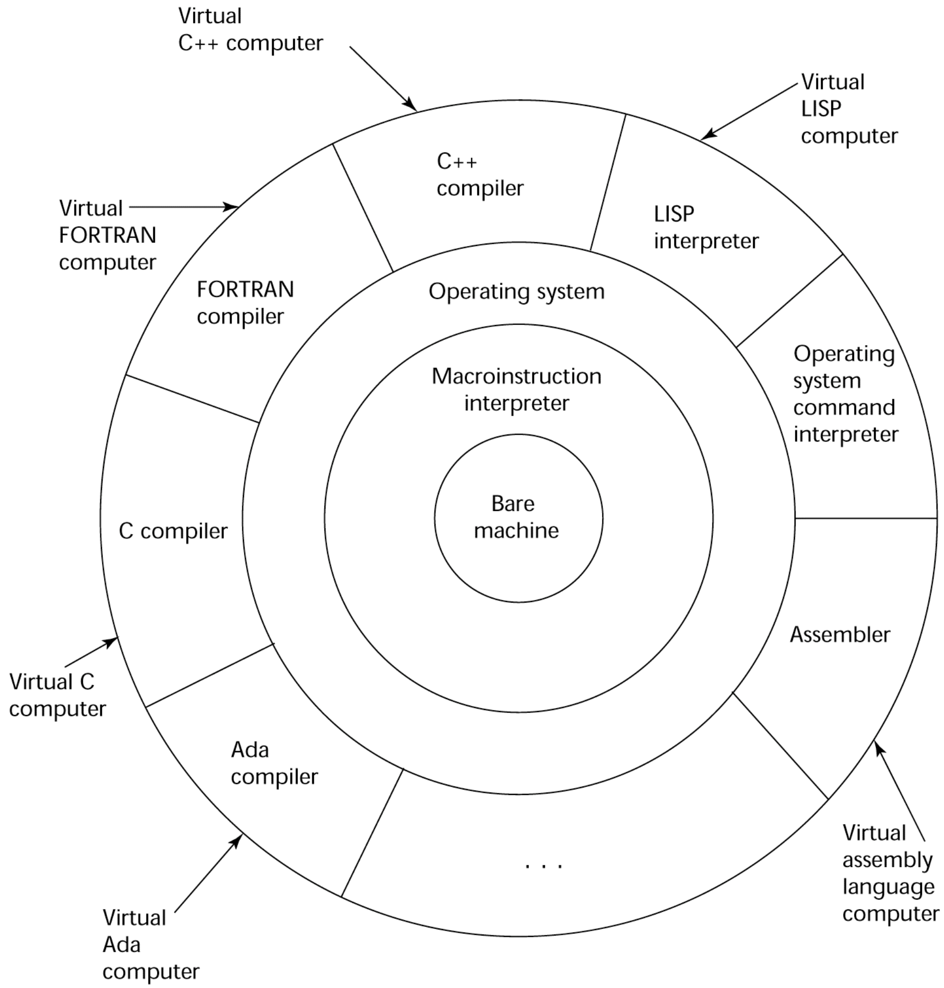
- Programming methodologies thru time:
  - 1950s and early 1960s:
    - Simple applications; worry about machine efficiency
  - Late 1960s:
    - People efficiency became important;
    - readability, better control structures
    - Structured programming
    - Top-down design and step-wise refinement
  - Late 1970s: Process-oriented to data-oriented
    - data abstraction
  - Middle 1980s: Re-use, Moudularity
    - Object-oriented programming
  - Late 1990s: Portability, reliability, security
    - Java,C#

Some Programming Paradigms

- Imperative
  - Central features are variables, assignment statements, and iteration
  - Examples: FORTRAN, C, Pascal
- Functional
  - Main means of making computations is by applying functions to given parameters
  - Examples: LISP, Scheme
- Logic
  - Rule-based
  - Rules are specified in no special order
  - Examples: Prolog
- Object-oriented
  - Encapsulate data objects with processing
  - Inheritance and dynamic type binding
  - Grew out of imperative languages
  - Examples: C++, Java

Languages typically support more than one paradigm although not equally well

Layered View of Computer

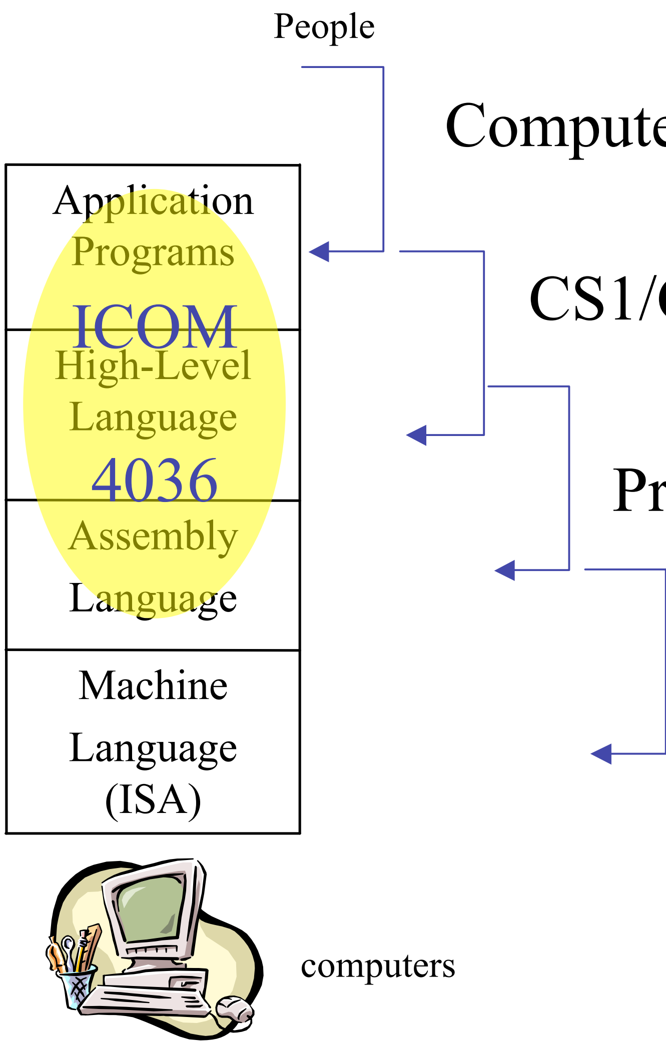


Each Layer Implements a Virtual Machine
with its own Programming Language

Virtual Machines (VM's)

Type of Virtual Machine	Examples	Instruction Elements	Data Elements	Comments
Application Programs	Spreadsheet, Word Processor	Drag & Drop, GUI ops, macros	cells, paragraphs, sections	Visual, Graphical, Interactive Application Specific Abstractions Easy for Humans Hides HLL Level
High-Level Language	C, C++, Java, FORTRAN, Pascal	if-then-else, procedures, loops	arrays, structures	Modular, Structured, Model Human Language/Thought General Purpose Abstractions Hides Lower Levels
Assembly-Level	SPIM, MASM	directives, pseudo-instructions, macros	registers, labelled memory cells	Symbolic Instructions/Data Hides some machine details like alignment, address calculations Exposes Machine ISA
Machine-Level (ISA)	MIPS, Intel 80x86	load, store, add, branch	bits, binary addresses	Numeric, Binary Difficult for Humans

Computing in Perspective



Computer Human Interaction, User Interfaces

CS1/CS2, Programming, Data Structures

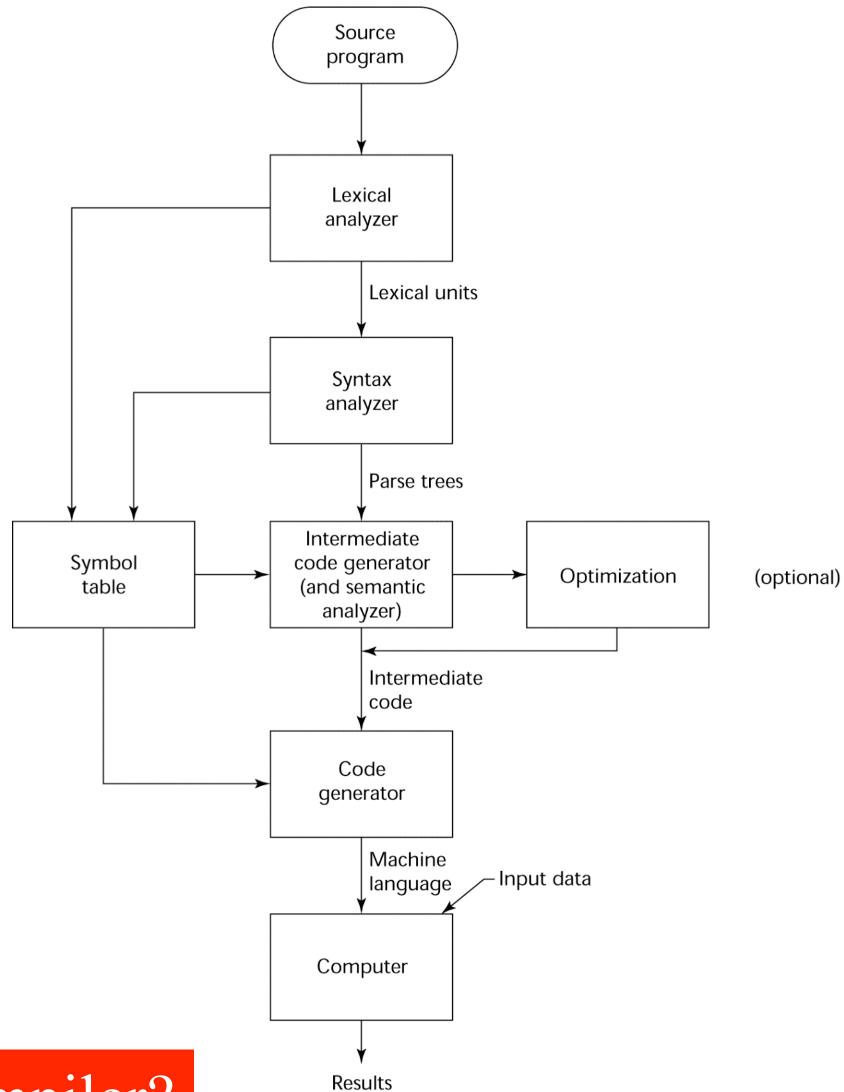
Programming Languages, Compilers

Computer Architecture

Each layer implements an
INTERPRETER
for some programming language

Implementation Methods
Compilation

- Translate high-level program to machine code
- Slow translation
- Fast execution



Trivia: Who developed the first compiler?

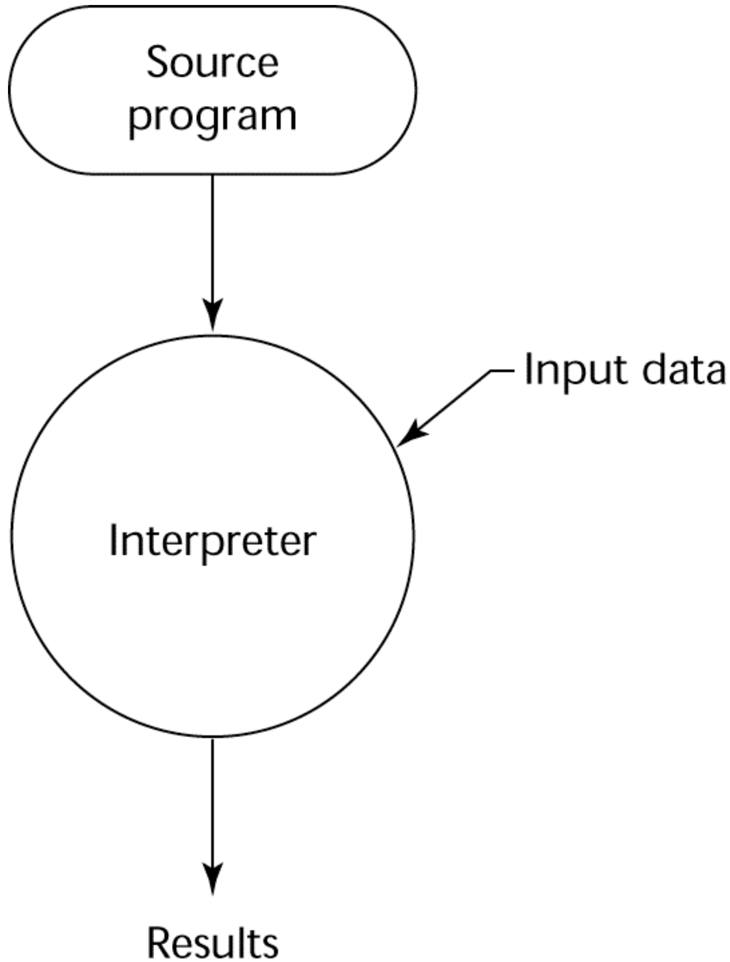
Answer: Computing Pioneer Grace Murray Hopper developed the first compiler ever

1984 picture

Learn more about Grace Murray Hopper @ wikipedia.org

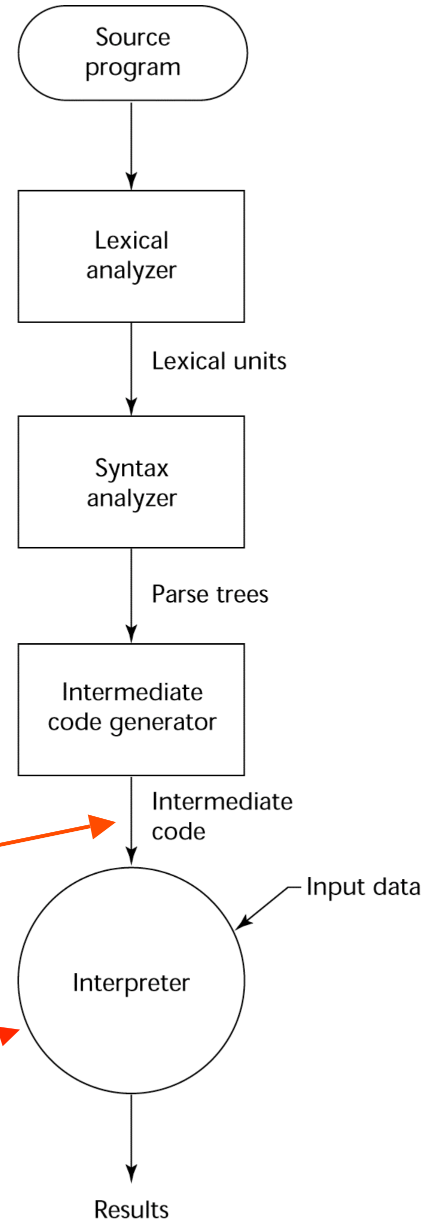
Implementation Methods
Interpretation

- No translation
- Slow execution
- Common in Scripting Languages



Implementation Methods
Hybrid Approaches

- Small translation cost
- Medium execution speed
- Portability



Examples of Intermediate Languages:

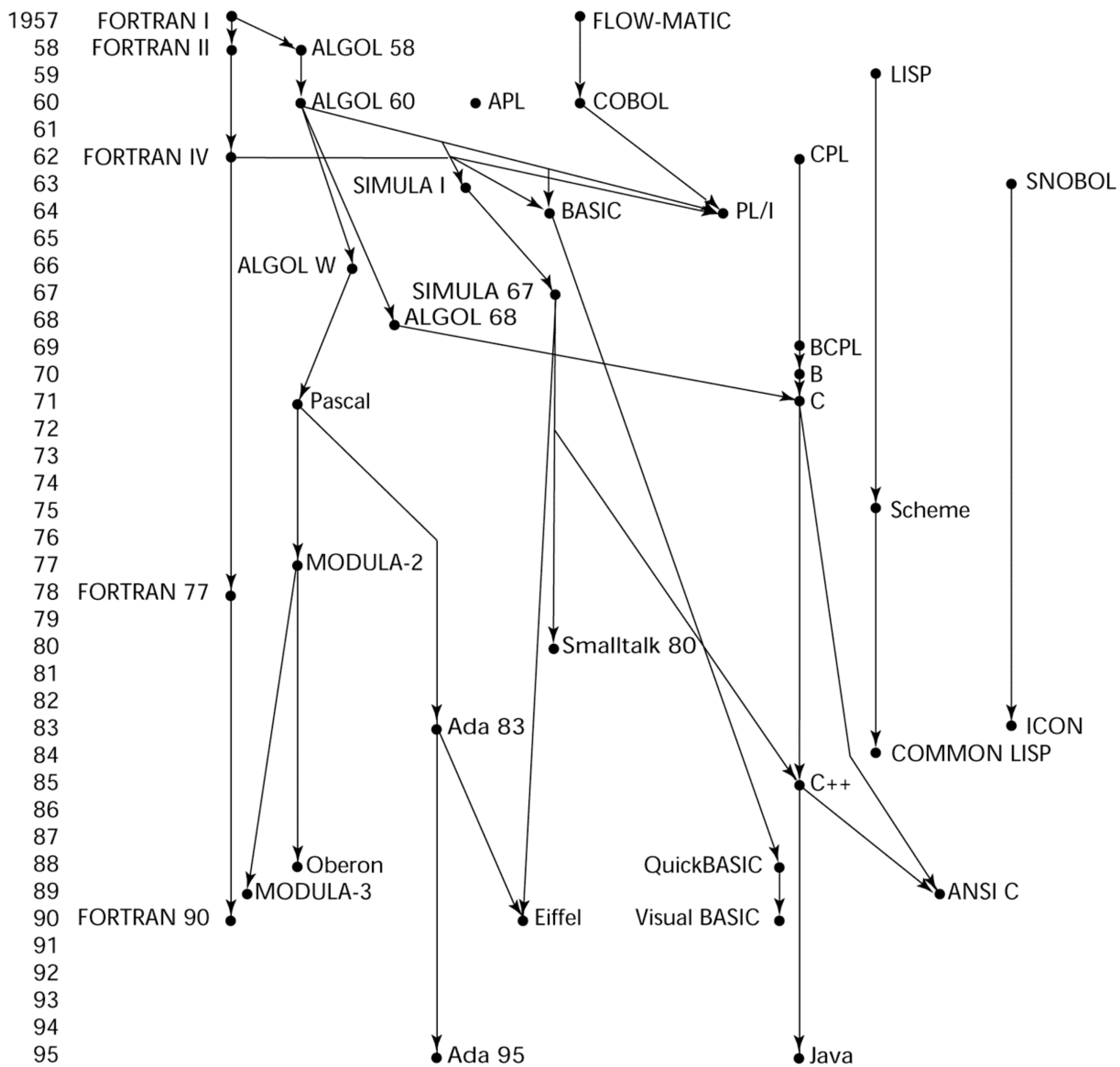
- Java Bytecodes
- .NET MSIL

Java VM

Software Development Environments (SDE's)

- The collection of tools used in software development
- GNU/FSF Tools
  - Emacs, GCC, GDB, Make
- Eclipse
  - An integrated development environment for Java
- Microsoft Visual Studio.NET
  - A large, complex visual environment
  - Used to program in C#, Visual BASIC.NET, Jscript, J#, or C++
- IBM WebSphere Studio
  - Specialized with many wizards to support webapp development

Genealogy of High-Level Languages



Machine Code – Computer's Native Language

- Binary encoded instruction sequence
- Architecture specific
- Interpreted by the processor
- Hard to read and debug

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a > 0) {
      a = a - b;
      result ++;
    }
  }
}
```

Address	I Bit	Opcode (binary)	X (base 10)
0	0	00 110	0
2	0	00 111	12
4	0	00 100	1000
6	0	00 110	0
8	[illegible]	[illegible]	4
10	[illegible]	[illegible]	1004
12	[illegible]	[illegible]	0
14	0	00 100	1008
16	0	00 101	1004
18	0	00 000	unused
20	0	00 111	1
22	1	00 111	1000
24	0	00 010	46
26	0	00 101	1000
28	0	00 010	46
30	0	00 101	1004
32	0	00 000	unused
34	0	00 111	1
36	0	00 100	1000
38	0	00 101	1008
40	0	00 111	1
42	0	00 100	1008
44	0	00 011	26

Machine Code Instruction:
0001110000001100_2
$1C0C_{16}$

Assembly Language

Improvements

- Symbolic names for each machine instruction
- Symbolic addresses
- Macros

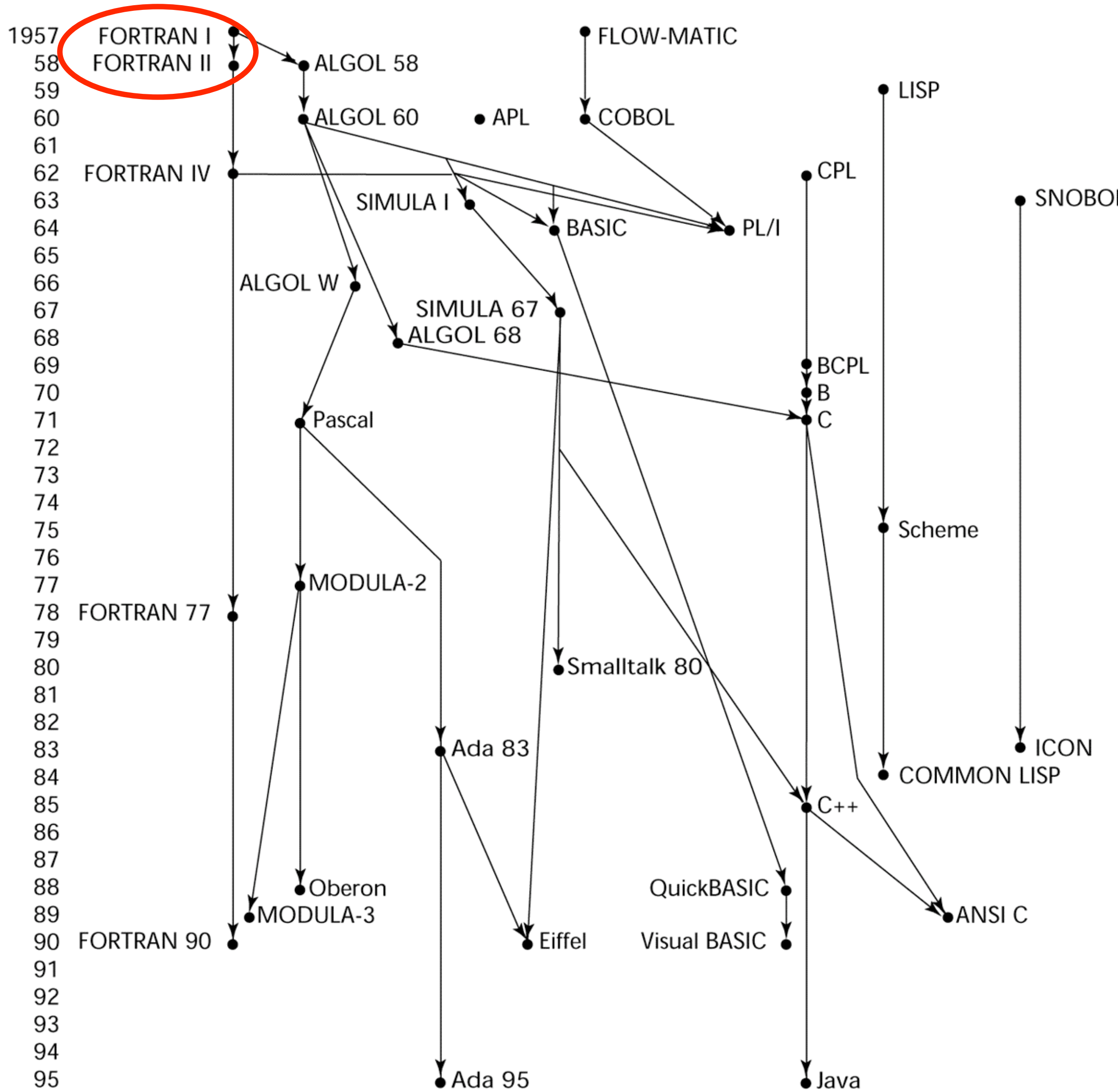
But

- Requires translation step
- Still architecture specific

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a > 0) {
      a = a - b;
      result ++;
    }
  }
}
```

```
0:        andi   0          # AC = 0
          addi   12
          storei 1000       # a = 12 (a stored @ 1000)
          andi   0          # AC = 0
          addi   4
          storei 1004       # b = 4  (b stored @ 1004)
          andi   0          # AC = 0
          storei 1008       # result = 0 (result @ 1008)
main:     loadi  1004       # compute a – b in AC
          comp              # using 2's complement add
          addi   1
          add    1000
          brni   exit       # exit if AC negative
loop:     loadi  1000
          brni   endloop
          loadi  1004       # compute a – b in AC
          comp              # using 2's complement add
          addi   1
          add    1000       # Uses indirect bit I = 1
          storei 1000
          loadi  1008       # result = result + 1
          addi   1
          storei 1008
          jumpi  loop
endloop:
exit:
```

Genealogy of High-Level Languages



IBM 704 and the FORmula TRANslation Language

- State of computing technology at the time
  - Computers were resource limited and unreliable
  - Applications were scientific
  - No programming methodology or tools
  - Machine efficiency was most important
  - Programs written in key-punched cards
- As a consequence
  - Little need for dynamic storage
  - Need good array handling and counting loops
  - No string handling, decimal arithmetic, or powerful input/output (commercial stuff)
  - Inflexible lexical/syntactic structure

FORTRAN Example

Some Improvements:

- Architecture independence
- Static Checking
- Algebraic syntax
- Functions/Procedures
- Arrays
- Better support for Structured Programming
- Device Independent I/O
- Formatted I/O

```
 subroutine checksum(buffer,length,sum32)

C       Calculate a 32-bit 1's complement checksum of the input buffer, adding
C       it to the value of sum32.  This algorithm assumes that the buffer
C       length is a multiple of 4 bytes.

C       a double precision value (which has at least 48 bits of precision)
C       is used to accumulate the checksum because standard Fortran does not
C       support an unsigned integer datatype.

C       buffer  - integer buffer to be summed
C       length  - number of bytes in the buffer (must be multiple of 4)
C       sum32   - double precision checksum value (The calculated checksum
C                 is added to the input value of sum32 to produce the
C                 output value of sum32)

        integer buffer(*),length,i,hibits
        double precision sum32,word32
        parameter (word32=4.294967296D+09)
C                 (word32 is equal to 2**32)

C       LENGTH must be less than 2**15, otherwise precision may be lost
C       in the sum
        if (length .gt. 32768)then
            print *, 'Error: size of block to sum is too large'
            return
        end if

        do i=1,length/4
            if (buffer(i) .ge. 0)then
                sum32=sum32+buffer(i)
            else
C               sign bit is set, so add the equivalent unsigned value
                sum32=sum32+(word32+buffer(i))
            end if
        end do

C       fold any overflow bits beyond 32 back into the word
10      hibits=sum32/word32
        if (hibits .gt. 0)then
            sum32=sum32-(hibits*word32)+hibits
            go to 10
        end if

        end
```

FORTRAN I (1957)

- First implemented version of FORTRAN
- Compiler released in April 1957 (18 worker-years of effort)
- Language Highlights
  - Names could have up to six characters
  - Post-test counting loop (`DO`)
  - Formatted I/O
  - User-defined subprograms
  - Three-way selection statement (arithmetic `IF`)
  - No data typing statements
  - No separate compilation
  - Code was very fast
  - Quickly became widely used

John W. Backus

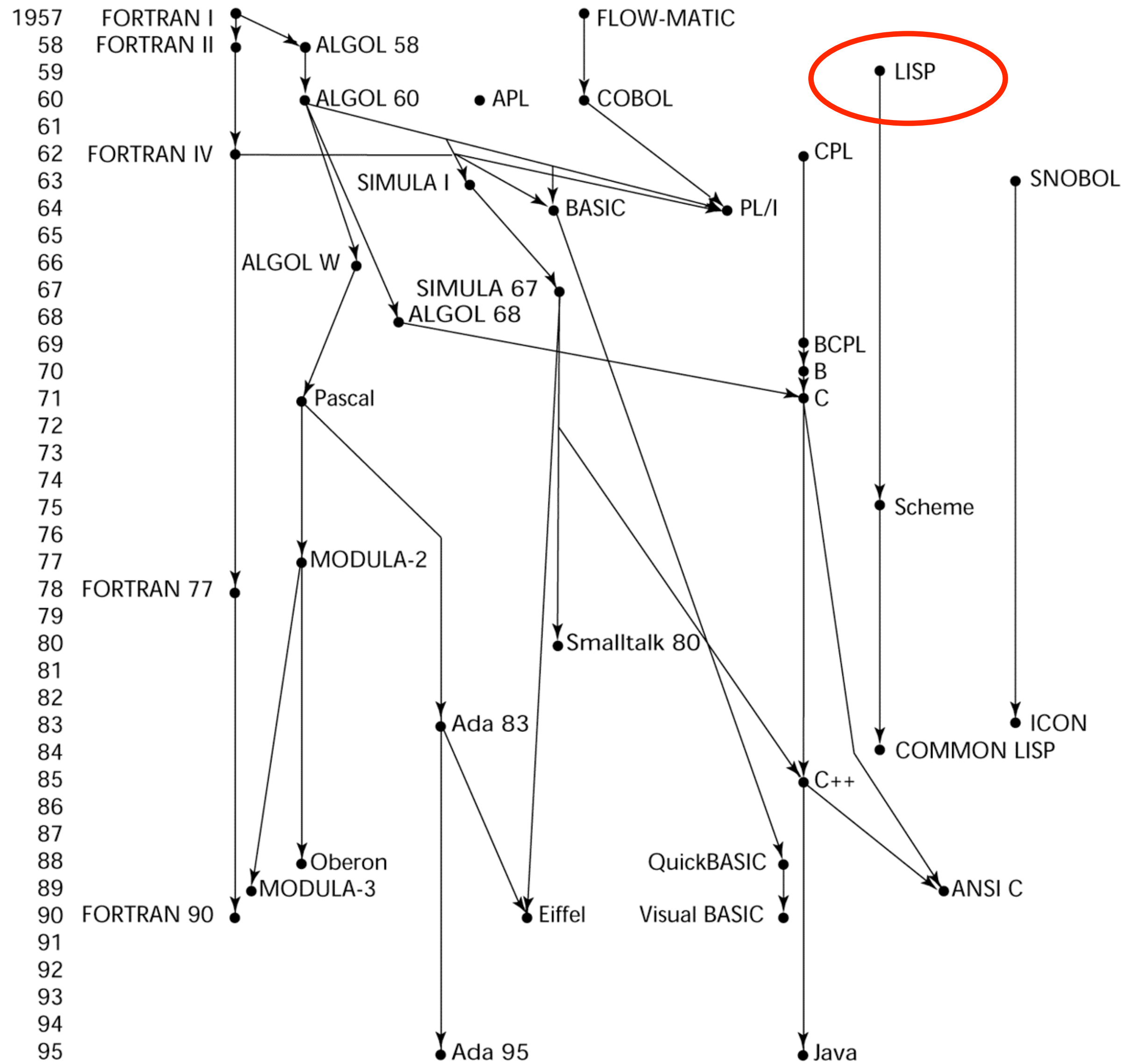
Many of these features are still dominant in current PLs

All Languages Evolve

Fifty years and still one of the most widely used languages in the planet!

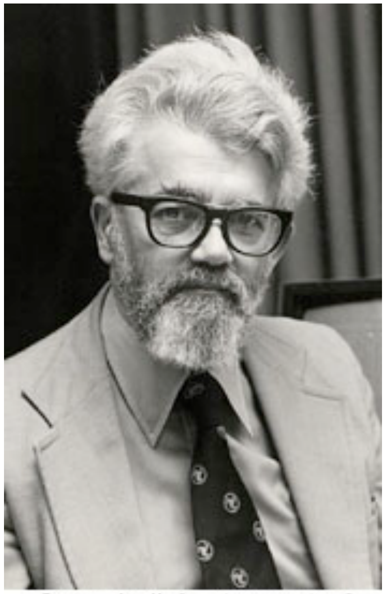
- FORTRAN 0 (1954)
- FORTRAN I (1957)
- FORTRAN II (1958)
  - Independent or separate compilation
  - Fixed compiler bugs
- FORTRAN IV (1960-62)
  - Explicit type declarations
  - Logical selection statement
  - Subprogram names could be parameters
  - ANSI standard in 1966
- FORTRAN 77 (1978)
  - Character string handling
  - Logical loop control statement
  - **`IF-THEN-ELSE`** statement
  - Still no recursion
- FORTRAN 90 (1990)
  - Modules
  - Dynamic arrays
  - Pointers
  - Recursion
  - **CASE** statement
  - Parameter type checking

Genealogy of High-Level Languages



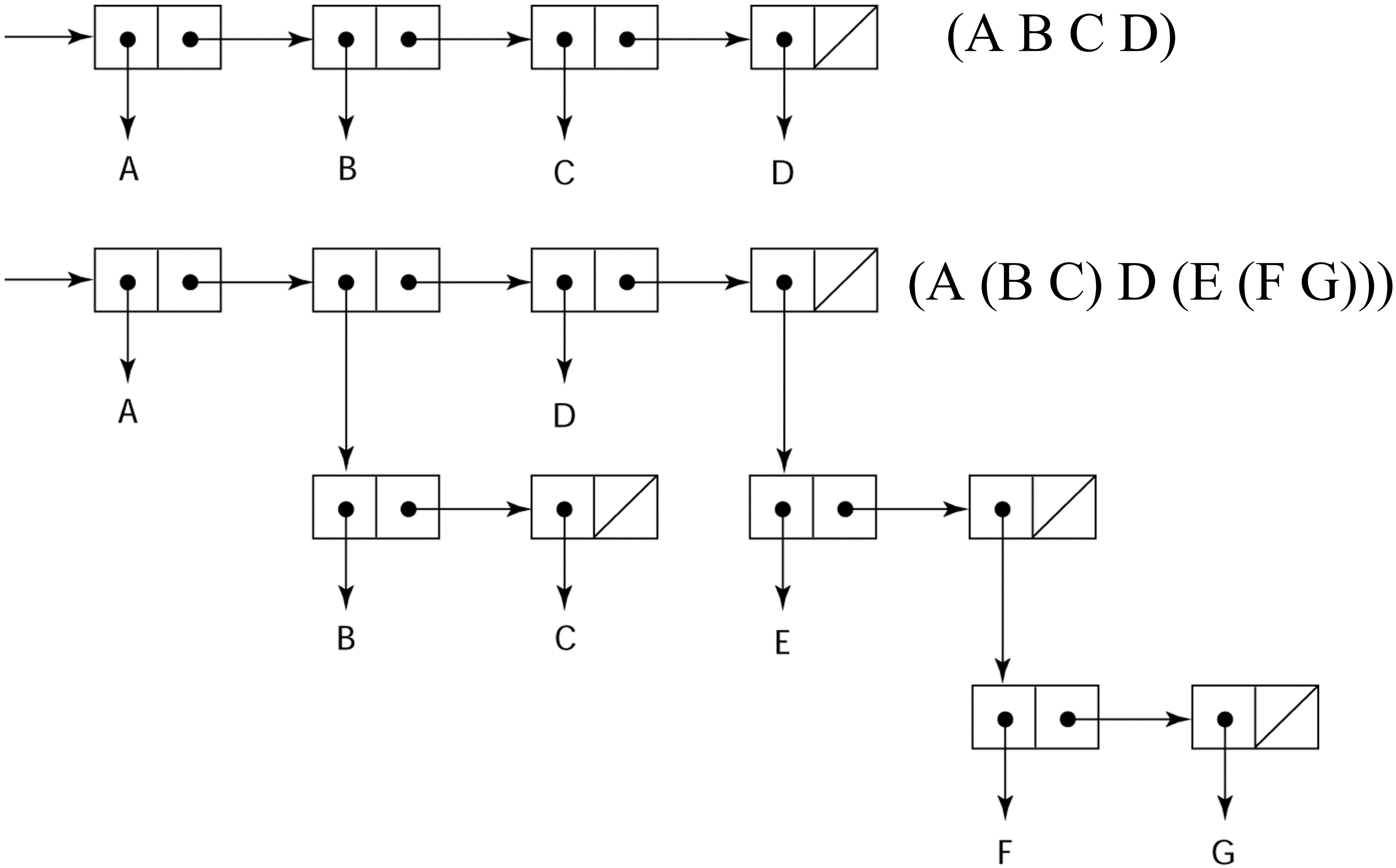
LISP - 1959

- LISt Processing language
  (Designed at MIT by McCarthy)
- AI research needed a language that:
  - Process data in lists (rather than arrays)
  - Symbolic computation (rather than numeric)
- Only two data types: atoms and lists
- Syntax is based on lambda calculus
- Pioneered functional programming
  - No need for variables or assignment
  - Control via recursion and conditional expressions
- Same syntax for data and code


●Lisp 研究の権威、John McCarthy 氏

The original LISP paper is here

Representation of Two LISP Lists

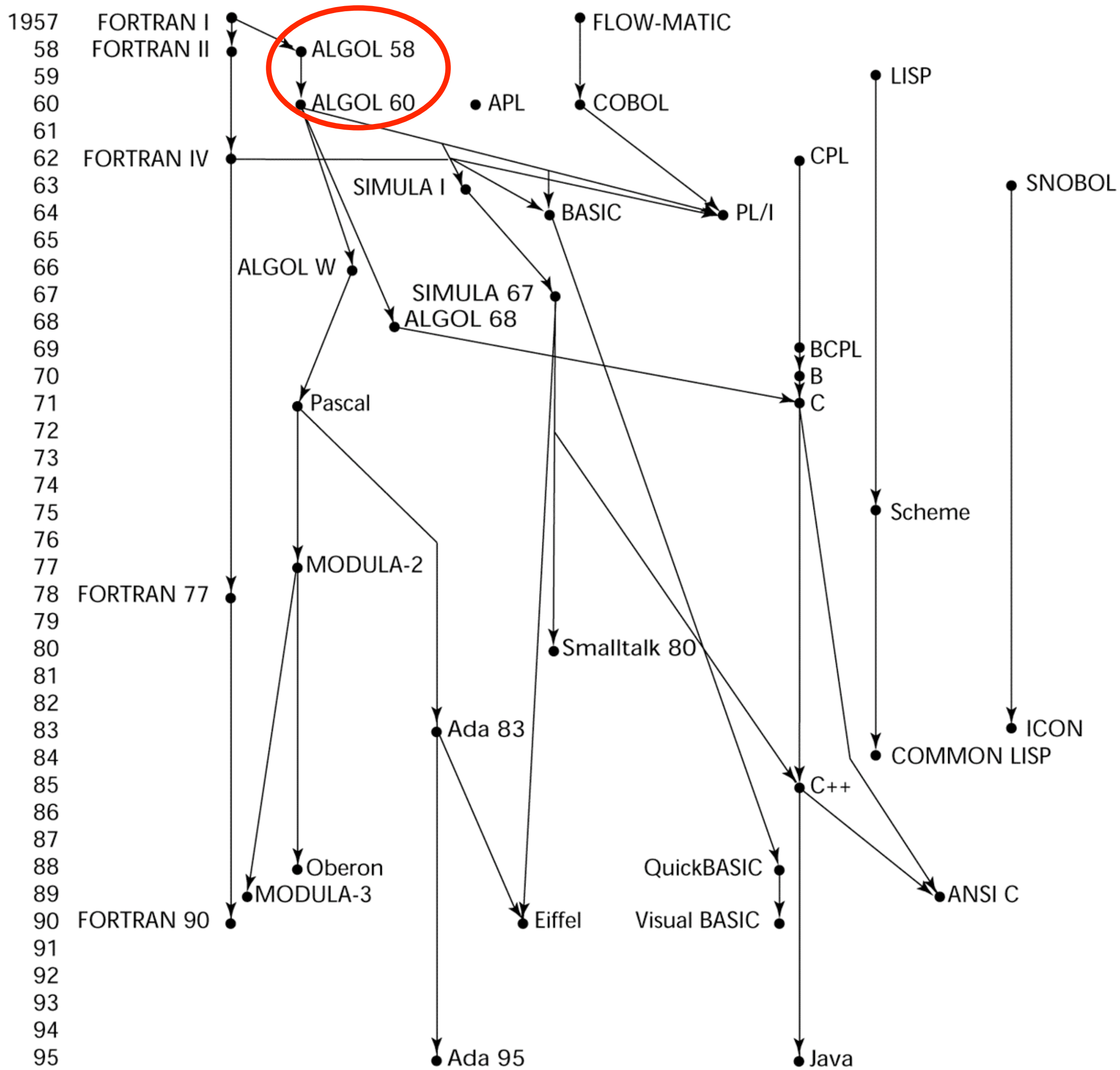


Scheme Example

```scheme
;;; From: Structure and Interpretation of Computer Programs
;;; (Harold Abelson and Gerald Jay Sussman with Julie Sussman)

;;; Added by Bjoern Hoefling (for usage with MIT-Scheme)

(define (atom? x)
  (or (number? x)
      (string? x)
      (symbol? x)
      (null? x)
      (eq? x #t)))

;;; Section 2.2.4 -- Symbolic differentiation

(define (deriv exp var)
  (cond ((constant? exp) 0)
        ((variable? exp)
         (if (same-variable? exp var) 1 0))
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var)))
        ((product? exp)
         (make-sum
          (make-product (multiplier exp)
                        (deriv (multiplicand exp) var))
          (make-product (deriv (multiplier exp) var)
                        (multiplicand exp))))))

(define (constant? x) (number? x))

(define (variable? x) (symbol? x))

(define (same-variable? v1 v2)
  (and (variable? v1) (variable? v2) (eq? v1 v2)))

(define (make-sum a1 a2) (list '+ a1 a2))

(define (make-product m1 m2) (list '* m1 m2))

(define (sum? x)
  (if (not (atom? x)) (eq? (car x) '+) nil))

(define (addend s) (cadr s))

(define (augend s) (caddr s))
```

```scheme
(define (product? x)
  (if (not (atom? x)) (eq? (car x) '*) nil))

(define (multiplier p) (cadr p))

(define (multiplicand p) (caddr p))

;;; examples from the textbook

(deriv '(+ x 3) 'x)
;Value 1: (+ 1 0)
(deriv '(* x y) 'y)
;Value 2: (+ (* x 1) (* 0 y))
(deriv '(* (* x y) (+ x 3)) 'x)
;Value 3: (+ (* (* x y) (+ 1 0)) (* (+ (* x 0) (* 1 y)) (+ x 3)))

;;; Better versions of make-sum and make-product

(define (make-sum a1 a2)
  (cond ((and (number? a1) (number? a2)) (+ a1 a2))
        ((number? a1) (if (= a1 0) a2 (list '+ a1 a2)))
        ((number? a2) (if (= a2 0) a1 (list '+ a1 a2)))
        (else (list '+ a1 a2))))

(define (make-product m1 m2)
  (cond ((and (number? m1) (number? m2)) (* m1 m2))
        ((number? m1)
         (cond ((= m1 0) 0)
               ((= m1 1) m2)
               (else (list '* m1 m2))))
        ((number? m2)
         (cond ((= m2 0) 0)
               ((= m2 1) m1)
               (else (list '* m1 m2))))
        (else (list '* m1 m2))))

;;; same examples as above

(deriv '(+ x 3) 'x)
;Value: 1
(deriv '(* x y) 'y)
;Value: x
(deriv '(* (* x y) (+ x 3)) 'x)
;Value 4: (+ (* x y) (* y (+ x 3)))
```

Genealogy of High-Level Languages



ALGOL 58 and 60

- State of Affairs
  - FORTRAN had (barely) arrived for IBM 70x
  - Many other languages were being developed, all for specific machines
  - No portable language; all were machine-dependent
  - No universal language for communicating algorithms
- ACM and GAMM met for four days for design
- Goals of the language:
  - Close to mathematical notation
  - Good for describing algorithms
  - Must be translatable to machine code

ALGOL 58

- New language features:
  - Concept of type was formalized
  - Names could have any length
  - Arrays could have any number of subscripts
  - Parameters were separated by mode (in & out)
  - Subscripts were placed in brackets
  - Compound statements (**`begin ... end`**)
  - Semicolon as a statement separator. Free format syntax.
  - Assignment operator was :=
  - **`if`** had an **`else-if`** clause
  - No I/O - "would make it machine dependent"

ALGOL 60

- Modified ALGOL 58 at 6-day meeting in Paris
- New language features:
  - Block structure (local scope)
  - Two parameter passing methods
  - Subprogram recursion
  - Stack-dynamic arrays
  - Still no I/O and no string handling
- Successes:
  - It was the standard way to publish algorithms for over 20 years
  - All subsequent imperative languages are based on it
  - First machine-independent language
  - First language whose syntax was formally defined (BNF)

ALGOL 60

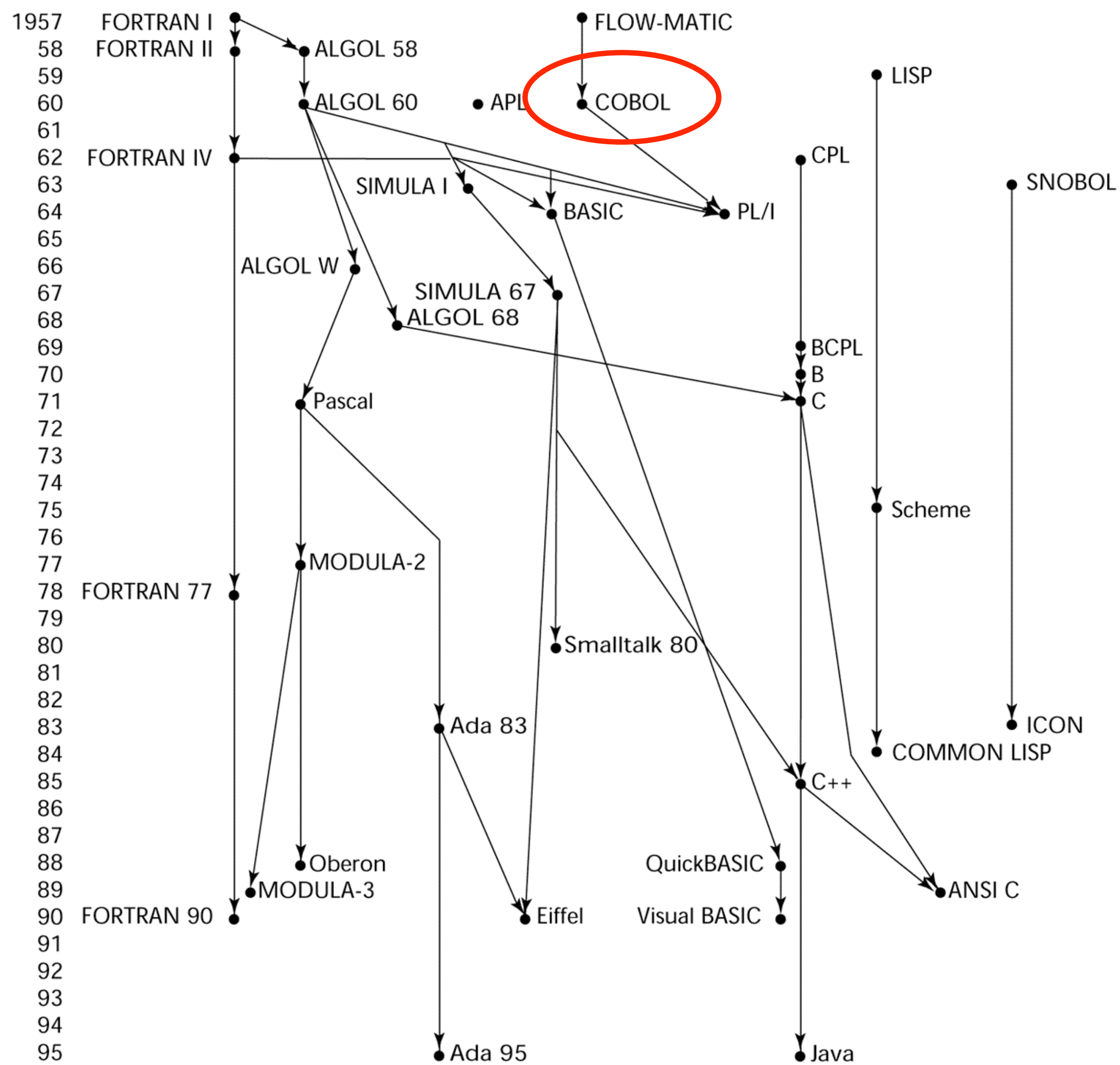
- Failure:
  - Never widely used, especially in U.S.
- Possible Reasons:
  - No I/O and the character set made programs non-portable
  - Too flexible--hard to implement
  - Entrenchment of FORTRAN
  - Formal syntax description
  - Lack of support of IBM

Good isn’t always popular

Algol 60 Example

```
'begin'
     'comment'
          create some random numbers, print them and
          print the average.
     ;

     'integer' NN;

     NN := 20;

     'begin'
          'integer' i;
          'real' sum;

          vprint ("random numbers:");

          sum := 0;
          'for' i := 1 'step' 1 'until' NN 'do' 'begin'
               'real' x;
               x := rand;
               sum := sum + x;
               vprint (i, x)
          'end';

          vprint ("average is:", sum / NN)
     'end'
'end'
```

Genealogy of High-Level Languages



COBOL

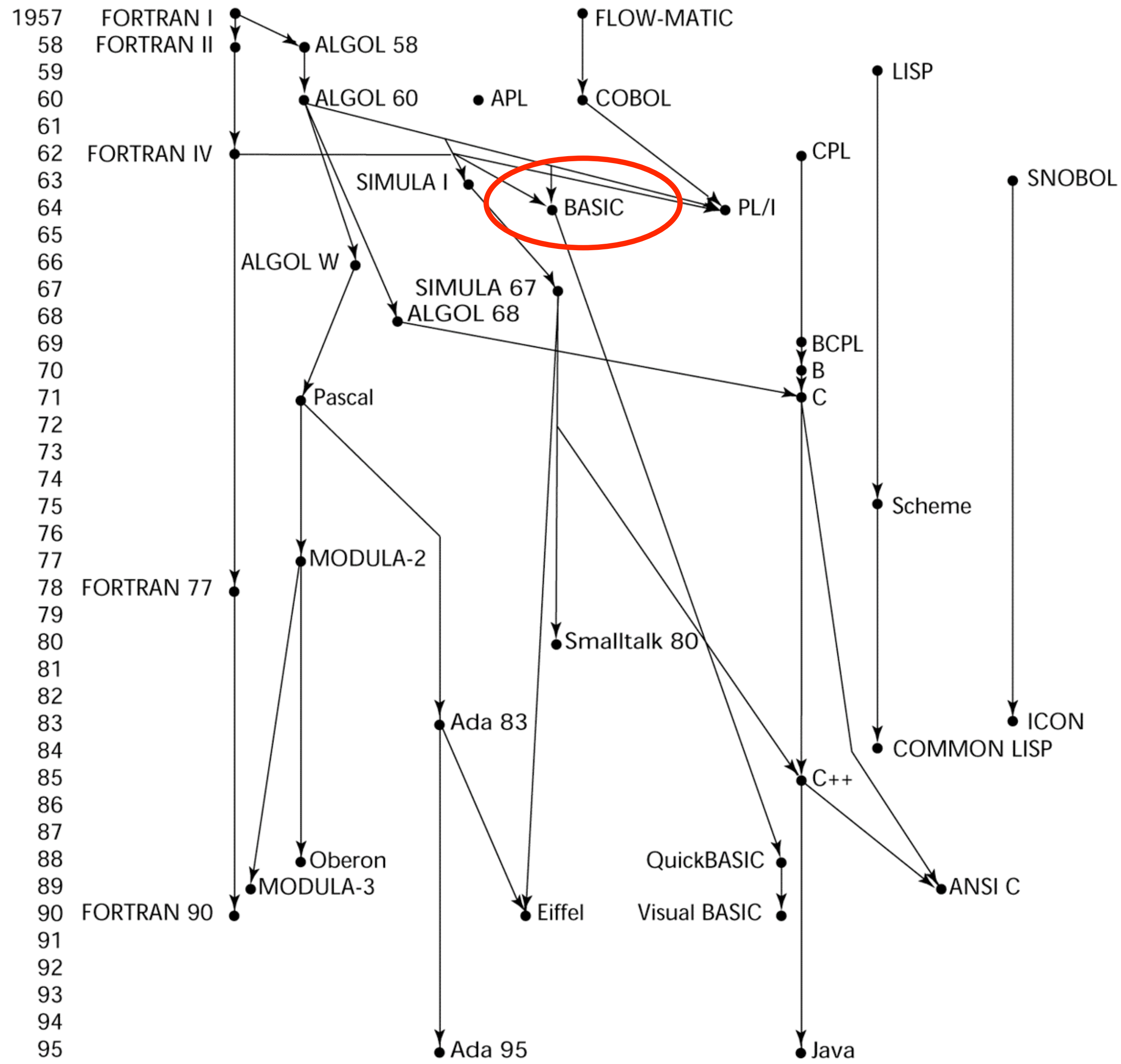
- Contributions:
  - First macro facility in a high-level language
  - Hierarchical data structures (records)
  - Nested selection statements
  - Long names (up to 30 characters), with hyphens
  - Separate data division
- Comments:
  - First language required by DoD
  - Still (2004) the most widely used business applications language

Cobol Example

```
$ SET SOURCEFORMAT"FREE"
IDENTIFICATION DIVISION.
PROGRAM-ID.  Iteration-If.
AUTHOR.  Michael Coughlan.

DATA DIVISION.
WORKING-STORAGE SECTION.
01  Num1           PIC 9  VALUE ZEROS.
01  Num2           PIC 9  VALUE ZEROS.
01  Result         PIC 99 VALUE ZEROS.
01  Operator       PIC X  VALUE SPACE.

PROCEDURE DIVISION.
Calculator.
   PERFORM 3 TIMES
      DISPLAY "Enter First Number      : " WITH NO ADVANCING
      ACCEPT Num1
      DISPLAY "Enter Second Number     : " WITH NO ADVANCING
      ACCEPT Num2
      DISPLAY "Enter operator (+ or *) : " WITH NO ADVANCING
      ACCEPT Operator
      IF Operator = "+" THEN
        ADD Num1, Num2 GIVING Result
      END-IF
      IF Operator = "*" THEN
        MULTIPLY Num1 BY Num2 GIVING Result
      END-IF
      DISPLAY "Result is = ", Result
   END-PERFORM.
   STOP RUN.
```

http://www.csis.ul.ie/COBOL/examples/

Genealogy of High-Level Languages



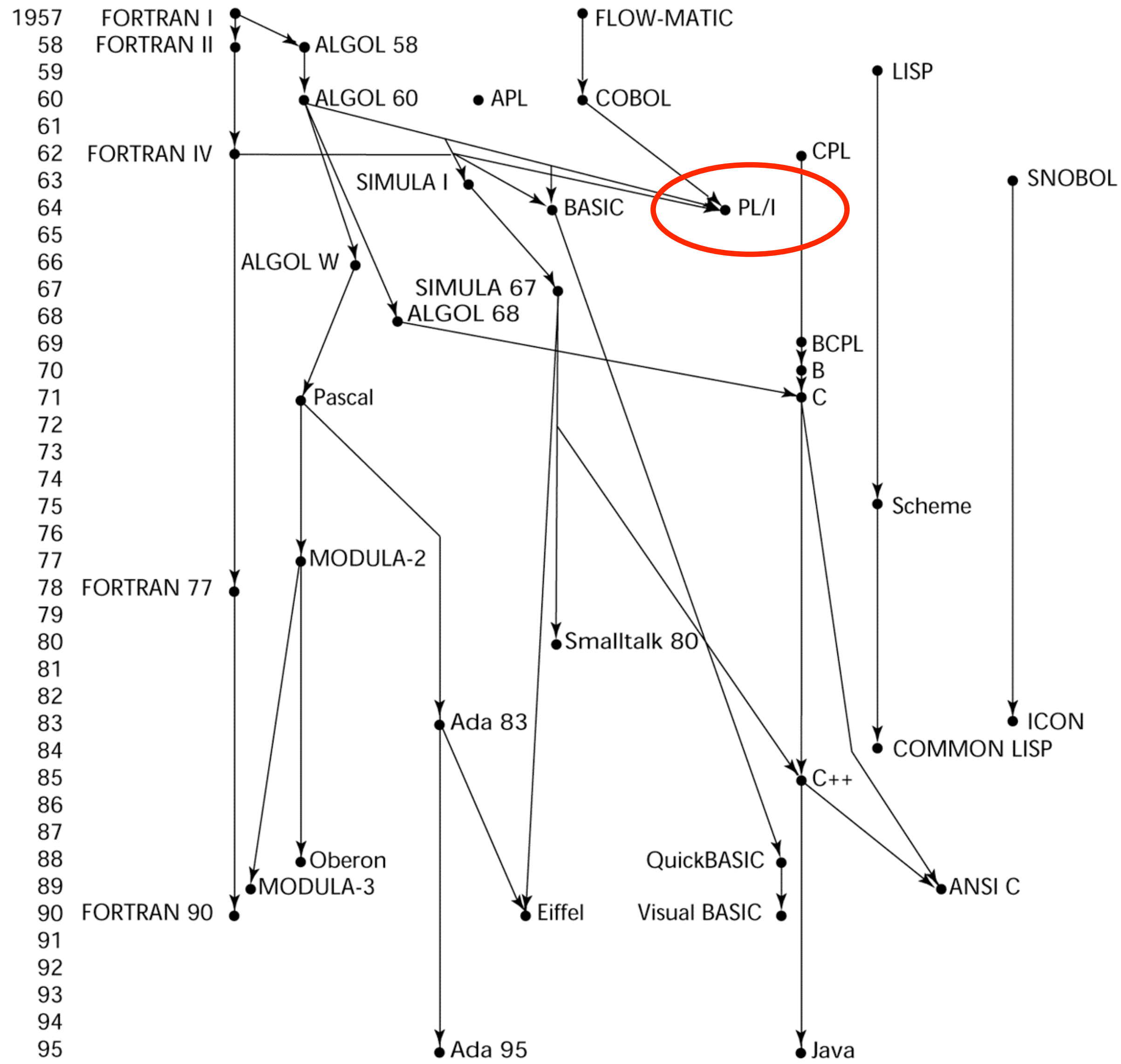
BASIC - 1964

- Designed by Kemeny & Kurtz at Dartmouth
- Design Goals:
  - Easy to learn and use for non-science students
  - Must be "pleasant and friendly"
  - Fast turnaround for homework
  - Free and private access
  - User time is more important than computer time
- Current popular dialect: Visual BASIC
- First widely used language with time sharing

Basic Example

```
1 DIM A(9)
10 PRINT "          TIC-TAC-TOE"
20 PRINT
30 PRINT "WE NUMBER THE SQUARES LIKE THIS:"
40 PRINT
50 PRINT 1,2,3
55 PRINT: PRINT
60 PRINT 4,5,6
70 PRINT 7,8,9
75 PRINT
80 FOR I=1 TO 9
90 A(I)=0
95 NEXT I
97 C=0
100 IF RND (2)=1 THEN 150              (flip a coin for first move)
110 PRINT "I'LL GO FIRST THIS TIME"
120 C=1
125 A(5)=1                             (computer always takes
130 PRINT                                 the center)
135 GOSUB 1000
140 goto 170
150 print "YOU MOVE FIRST"
160 PRINT
170 INPUT "WHICH SPACE DO YOU WANT",B
180 IF A(B)=0 THEN 195
185 PRINT "ILLEGAL MOVE"
190 GOTO 170
195 C=C+1                              (C is the move counter)
200 A(B)=1
205 GOSUB 1700
209 IF G=0 THEN 270                    (G is the flag signaling
211 IF C=9 THEN 260                       a win)
213 GOSUB 1500
215 C=C+1
220 GOSUB 1000
230 GOSUB 1700
235 IF G=0 THEN 270
250 IF C&lt;9 THEN 170
260 PRINT "TIE GAME!!!!"
265 PRINT
270 INPUT "PLAY GAIN (Y OR N)",A$
275 IF A$="Y" THEN 80                  (No need to Dimension a string
280 PRINT "SO LONG"                    with lengh of one)
285 END
995 REM *PRINT THE BOARD*
1000 FOR J=1 TO 3
1010 TAB 6
1020 PRINT "*";
1030 TAB 12
```

Genealogy of High-Level Languages



PL/I - 1965

- Designed by IBM and SHARE
- Computing situation in 1964 (IBM's point of view)
  - Scientific computing
    - IBM 1620 and 7090 computers
    - FORTRAN
    - SHARE user group
  - Business computing
    - IBM 1401, 7080 computers
    - COBOL
    - GUIDE user group
  - Compilers expensive and hard to maintain

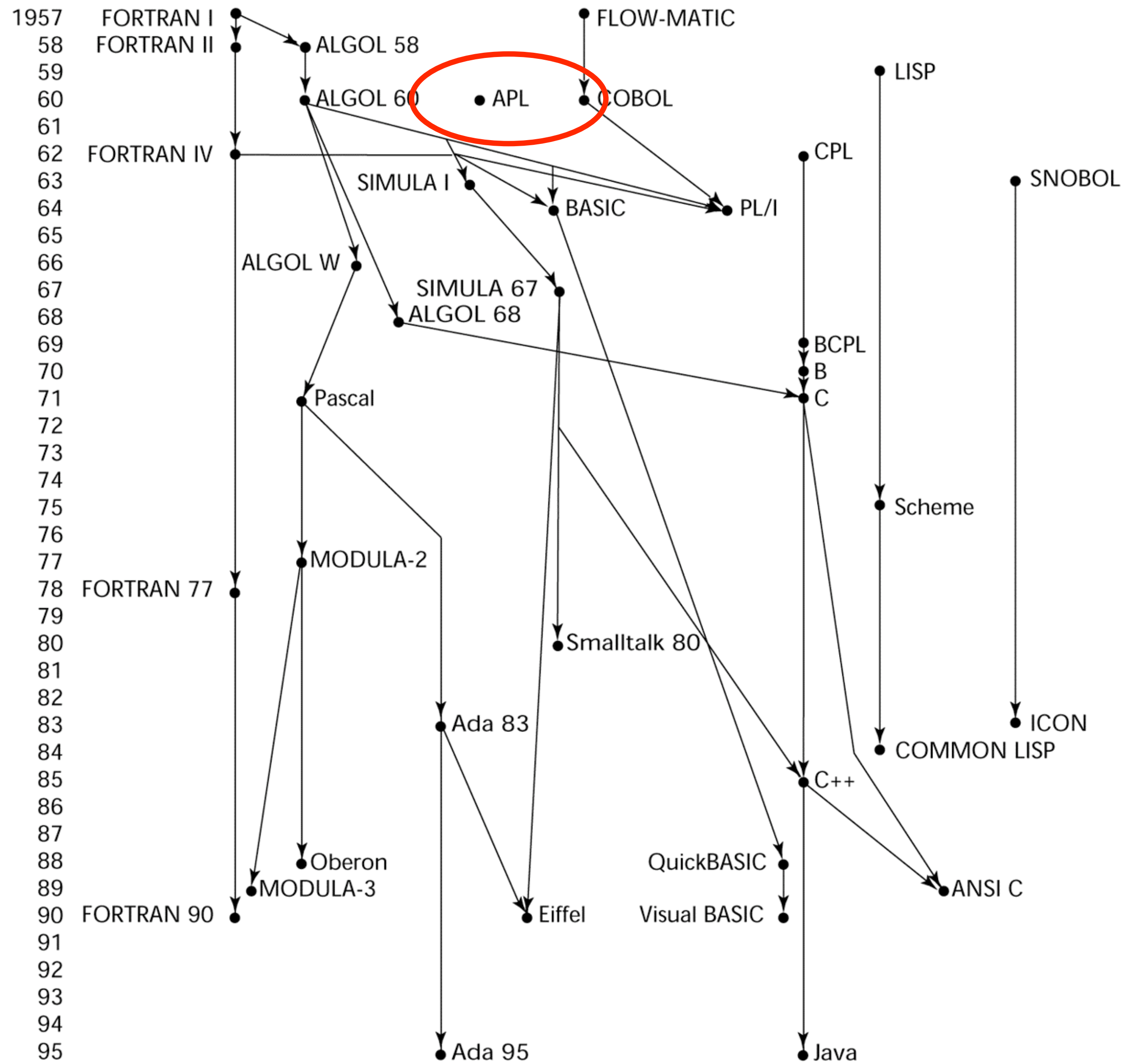
PL/I

- By 1963, however,
  – Scientific users began to need more elaborate I/O, like COBOL had; Business users began to need floating point and arrays (MIS)
  – It looked like many shops would begin to need two kinds of computers, languages, and support staff--too costly
- The obvious solution:
  – Build a new computer to do both kinds of applications
  – Design a new language to do both kinds of applications

PL/I

- Designed in five months by the 3 X 3 Committee
- PL/I contributions:
  - First unit-level concurrency
  - First exception handling
  - Switch-selectable recursion
  - First pointer data type
  - First array cross sections
- Comments:
  - Many new features were poorly designed
  - Too large and too complex
  - Was (and still is) actually used for both scientific and business applications

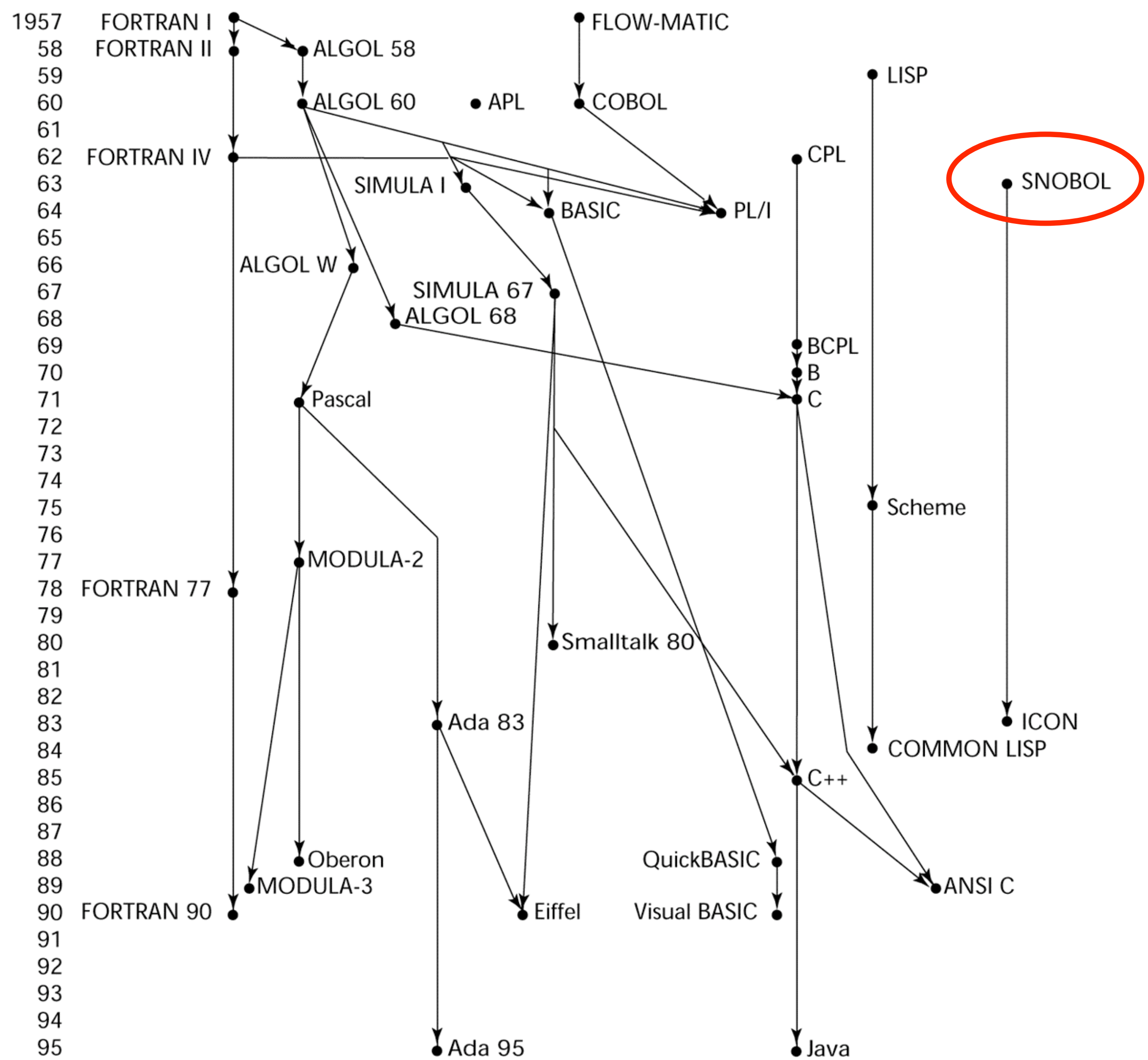
Genealogy of High-Level Languages



APL (1962)

- Characterized by dynamic typing and dynamic storage allocation
- APL (A Programming Language) 1962
  - Designed as a hardware description language (at IBM by Ken Iverson)
  - Highly expressive (many operators, for both scalars and arrays of various dimensions)
  - Programs are very difficult to read

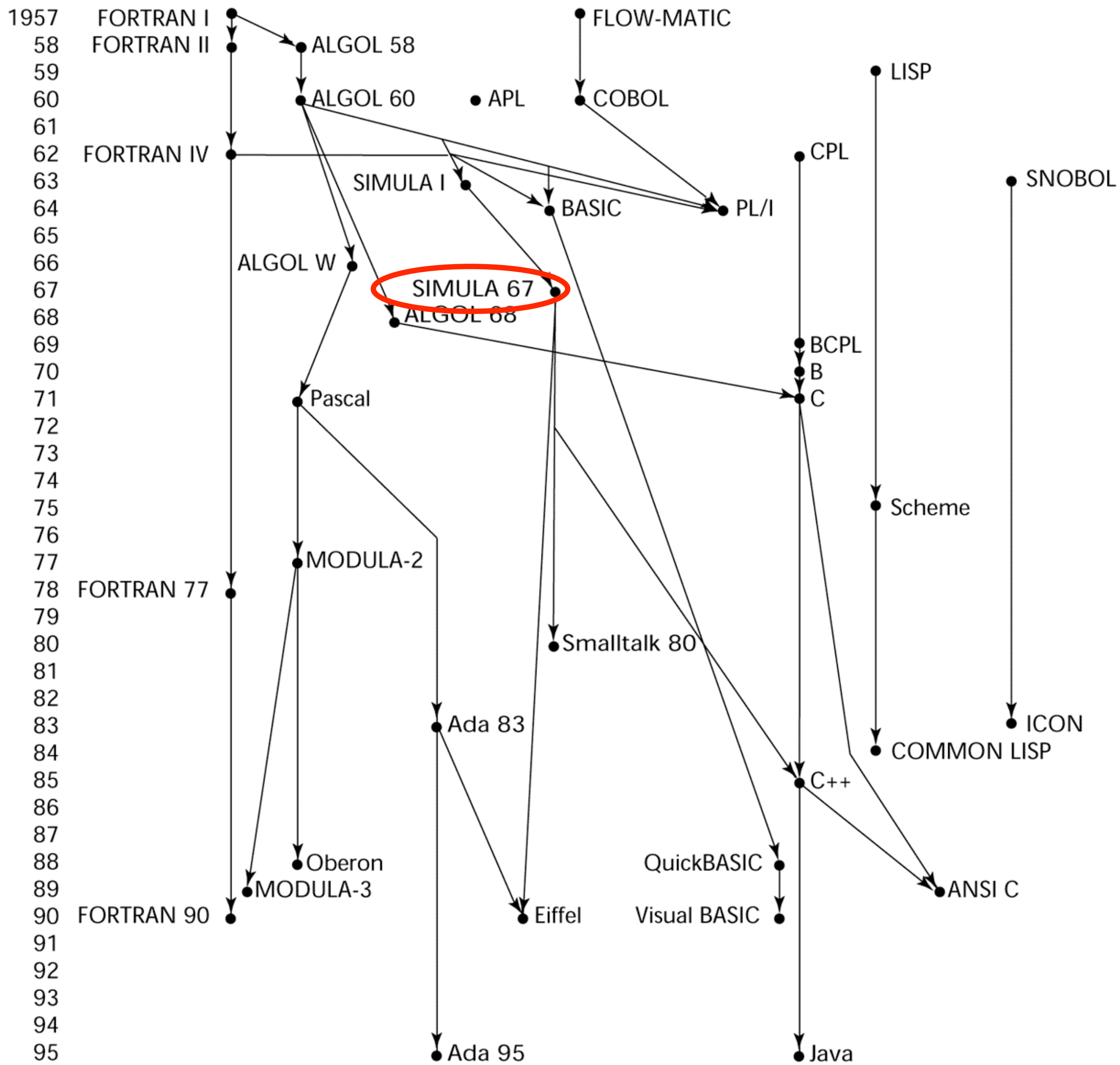
Genealogy of High-Level Languages



SNOBOL (1964)

- A string manipulation special purpose language
- Designed as language at Bell Labs by Farber, Griswold, and Polensky
- Powerful operators for string pattern matching

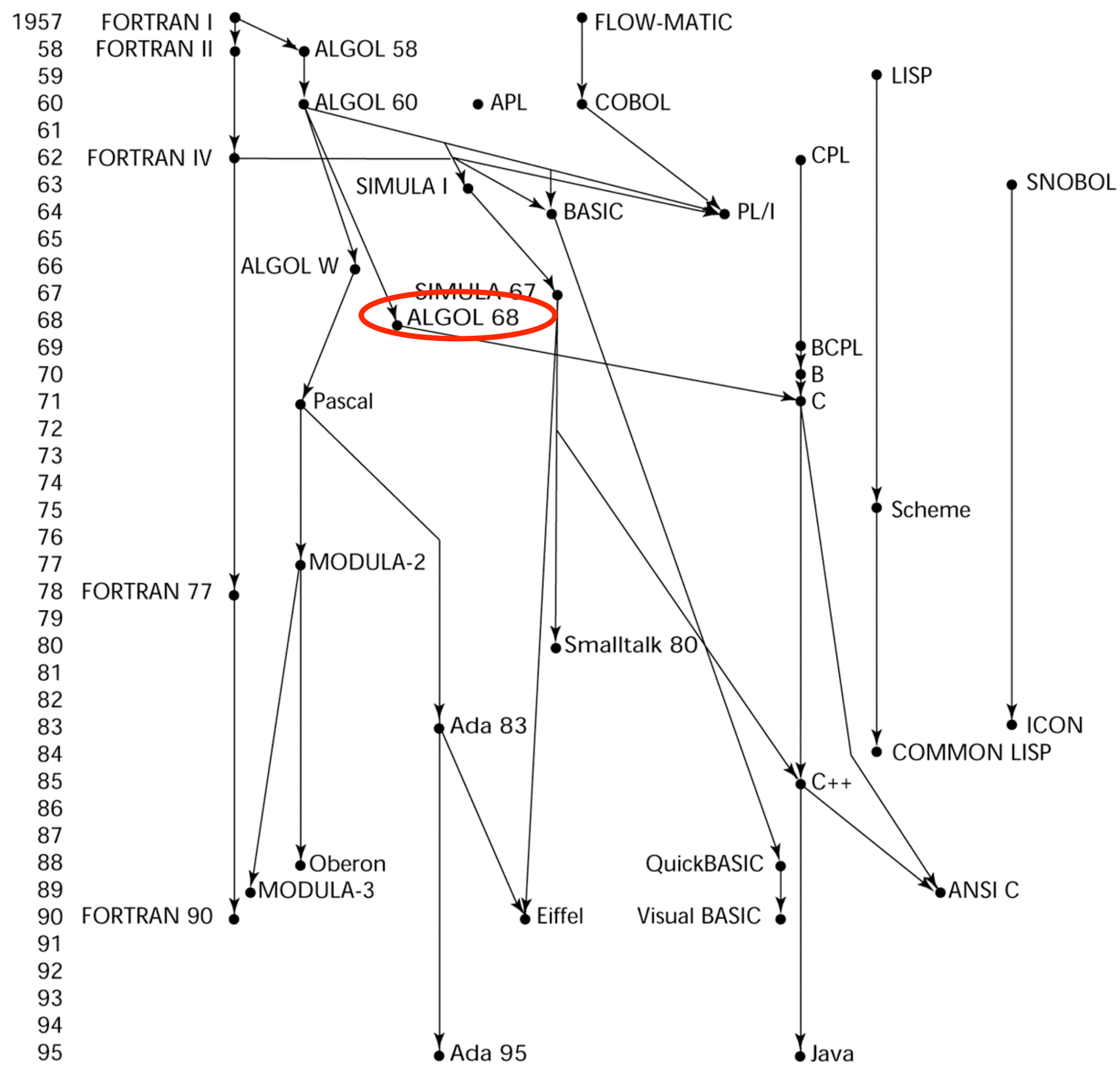
Genealogy of High-Level Languages



SIMULA 67 (1967)

- Designed primarily for system simulation (in Norway by Nygaard and Dahl)
- Based on ALGOL 60 and SIMULA I
- Primary Contribution:
  - Co-routines - a kind of subprogram
  - Implemented in a structure called a <u>class</u>
  - Classes are the basis for data abstraction
  - Classes are structures that include <u>both local data and functionality</u>
  - Supported <u>objects and inheritance</u>

Genealogy of High-Level Languages



ALGOL 68 (1968)

- Derived from, but not a superset of Algol 60
- Design goal is orthogonality
- Contributions:
  - User-defined data structures
  - Reference types
  - Dynamic arrays (called flex arrays)
- Comments:
  - Had even less usage than ALGOL 60
  - Had strong influence on subsequent languages, especially Pascal, C, and Ada

Important ALGOL Descendants I

- Pascal - 1971 (Wirth)
  - Designed by Wirth, who quit the ALGOL 68 committee (didn't like the direction of that work)
  - Designed for teaching structured programming
  - Small, simple, nothing really new
  - From mid-1970s until the late 1990s, it was the most widely used language for teaching programming in colleges

- C – 1972 (Dennis Richie)
  - Designed for systems programming
  - Evolved primarily from B, but also ALGOL 68
  - Powerful set of operators, but poor type checking
  - Initially spread through UNIX

Important ALGOL Descendants II

- Modula-2 - mid-1970s (Wirth)
  - Pascal plus modules and some low-level features designed for systems programming
- Modula-3 - late 1980s (Digital & Olivetti)
  - Modula-2 plus classes, exception handling, garbage collection, and concurrency
- Oberon - late 1980s (Wirth)
  - Adds support for OOP to Modula-2
  - Many Modula-2 features were deleted (e.g., **`for`** statement, enumeration types, **`with`** statement, noninteger array indices)

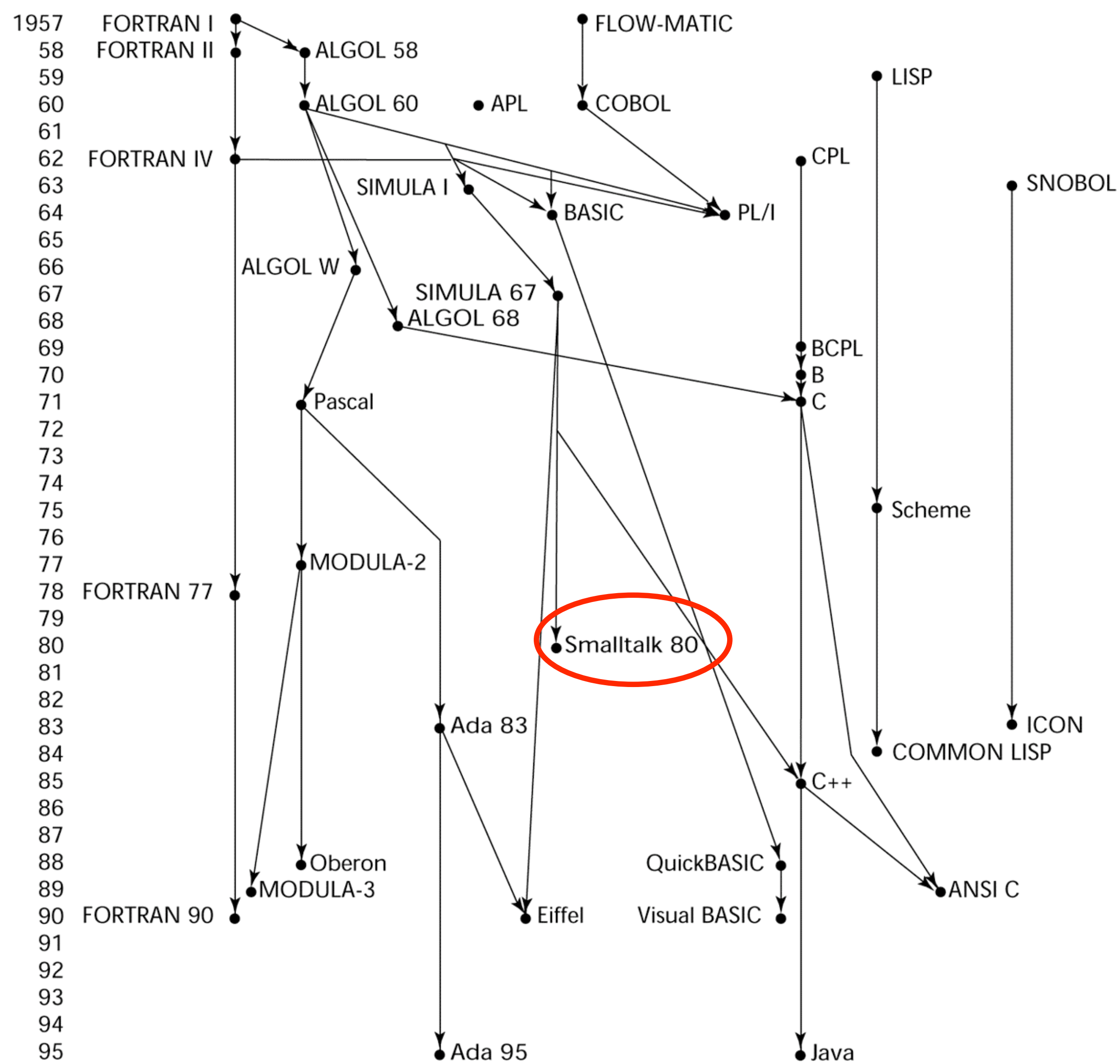
Prolog - 1972

- Developed at the University of Aix-Marseille, by Comerauer and Roussel, with some help from Kowalski at the University of Edinburgh
- Based on formal logic
- Non-procedural
- Can be summarized as being an intelligent database system that uses an inference process to infer the truth of given queries

Prolog Examples

```
fac1(0,1).
fac1(M,N) :- M1 is M-1, fac1(M1,N1), N is M*N1.


fac2(M,1) :- M =<0.
fac2(M,N) :- M1 is M-1, fac2(M1,N1), N is M*N1.


fac3(M,1) :- M =<0, !.
fac3(M,N) :- M1 is M-1, fac3(M1,N1), N is M*N1.
```

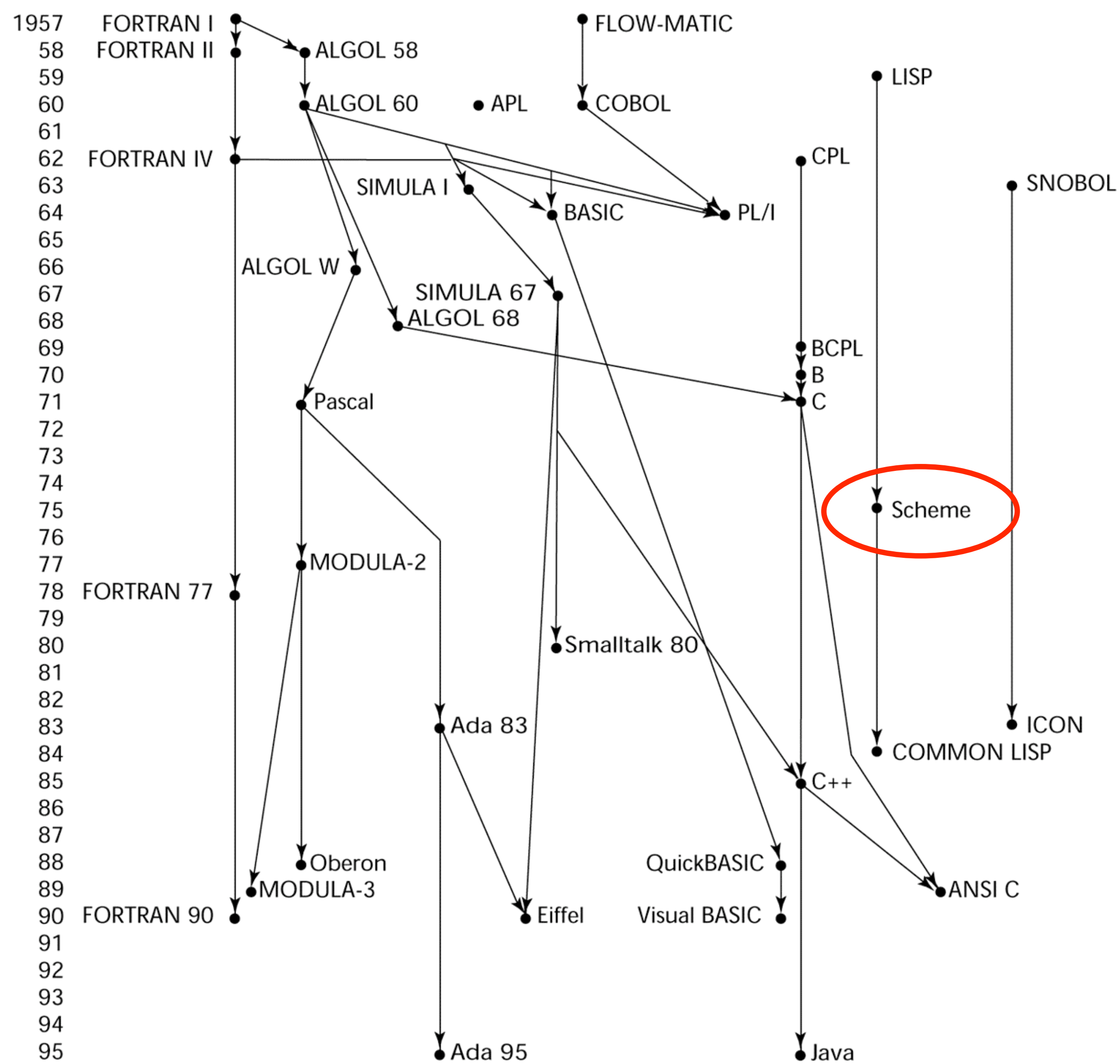
Genealogy of High-Level Languages



Smalltalk - 1972-1980

- Developed at Xerox PARC, initially by Alan Kay, later by Adele Goldberg
- First full implementation of an object-oriented language (data abstraction, inheritance, and dynamic type binding)
- Pioneered the graphical user interface everyone now uses

Genealogy of High-Level Languages



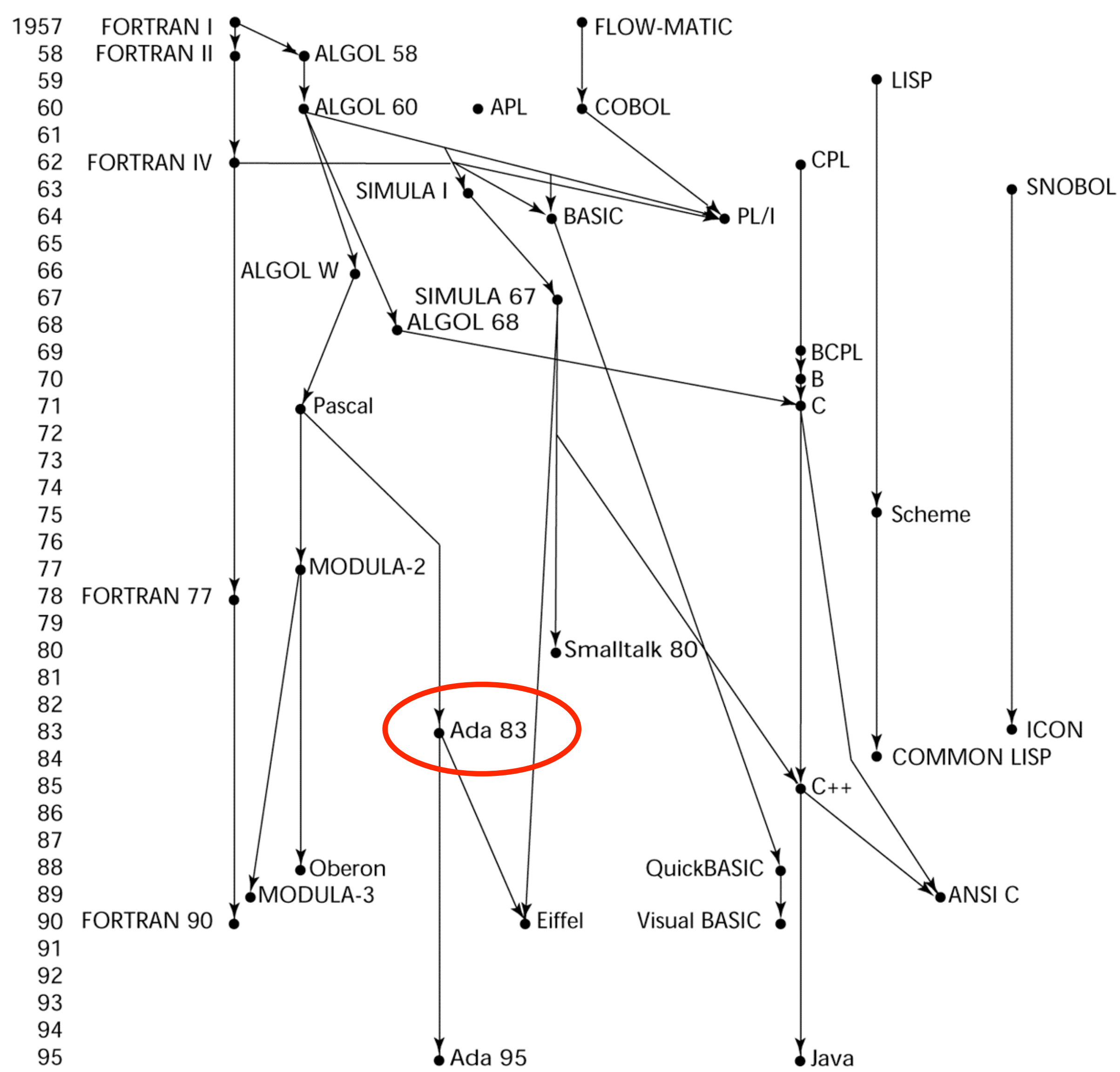
Scheme (1970's)

- MIT's dear programming language
- Designed by Gerald J. Sussman and Guy Steele Jr
- LISP with static scoping and closures
- Compiled code coexists with interpreted code
- Garbage collection
- Tail recursion
- Explicit Continuations

Sussman

Steele

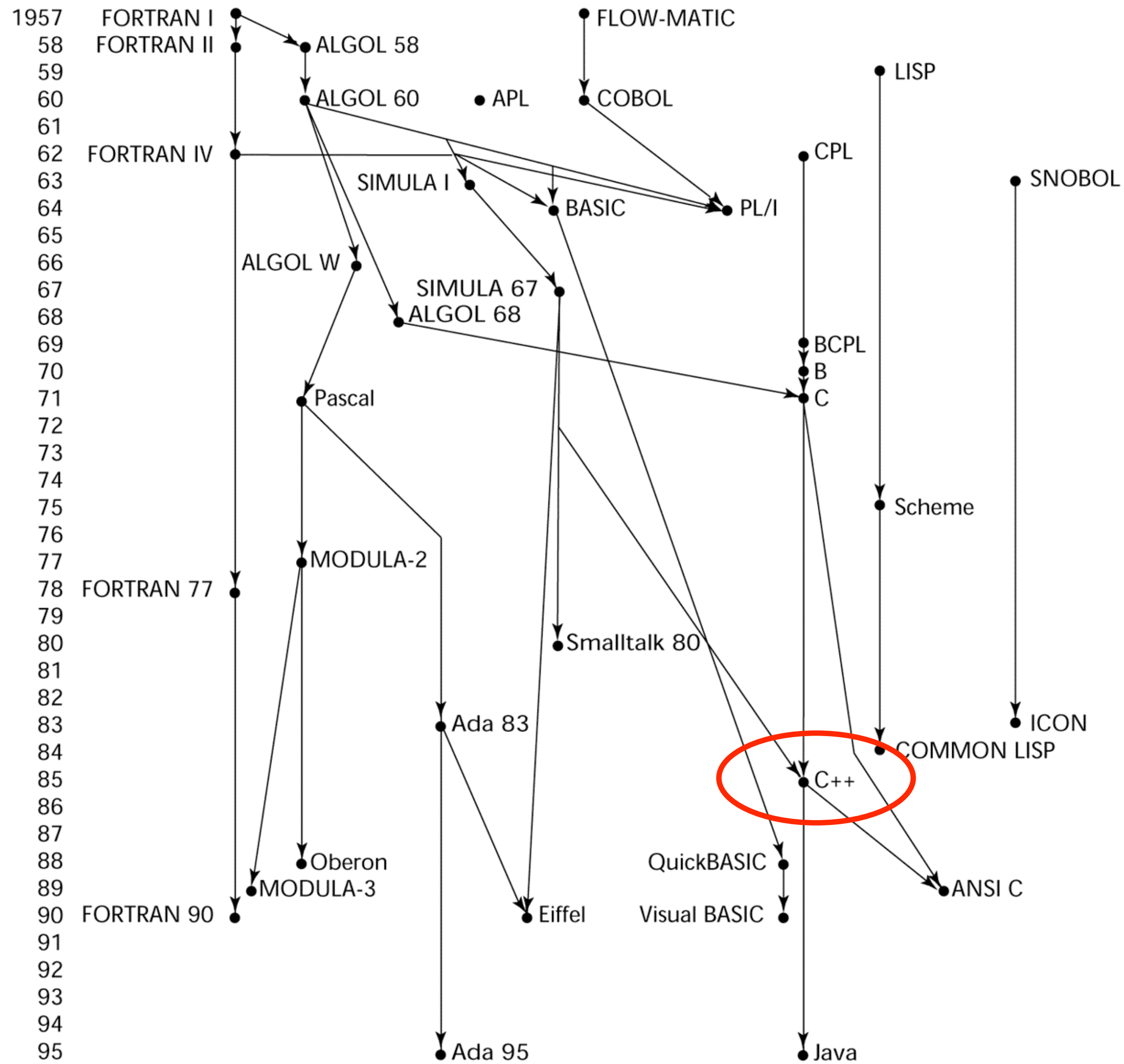
Genealogy of High-Level Languages



Ada - 1983 (began in mid-1970s)

- Huge design effort, involving hundreds of people, much money, and about eight years
- Environment: More than 450 different languages being used for DOD embedded systems (no software reuse and no development tools)
- Contributions:
  - Packages - support for data abstraction
  - Exception handling - elaborate
  - Generic program units
  - Concurrency - through the tasking model
- Comments:
  - Competitive design
  - Included all that was then known about software engineering and language design
  - First compilers were very difficult; the first really usable compiler came nearly five years after the language design was completed

Genealogy of High-Level Languages



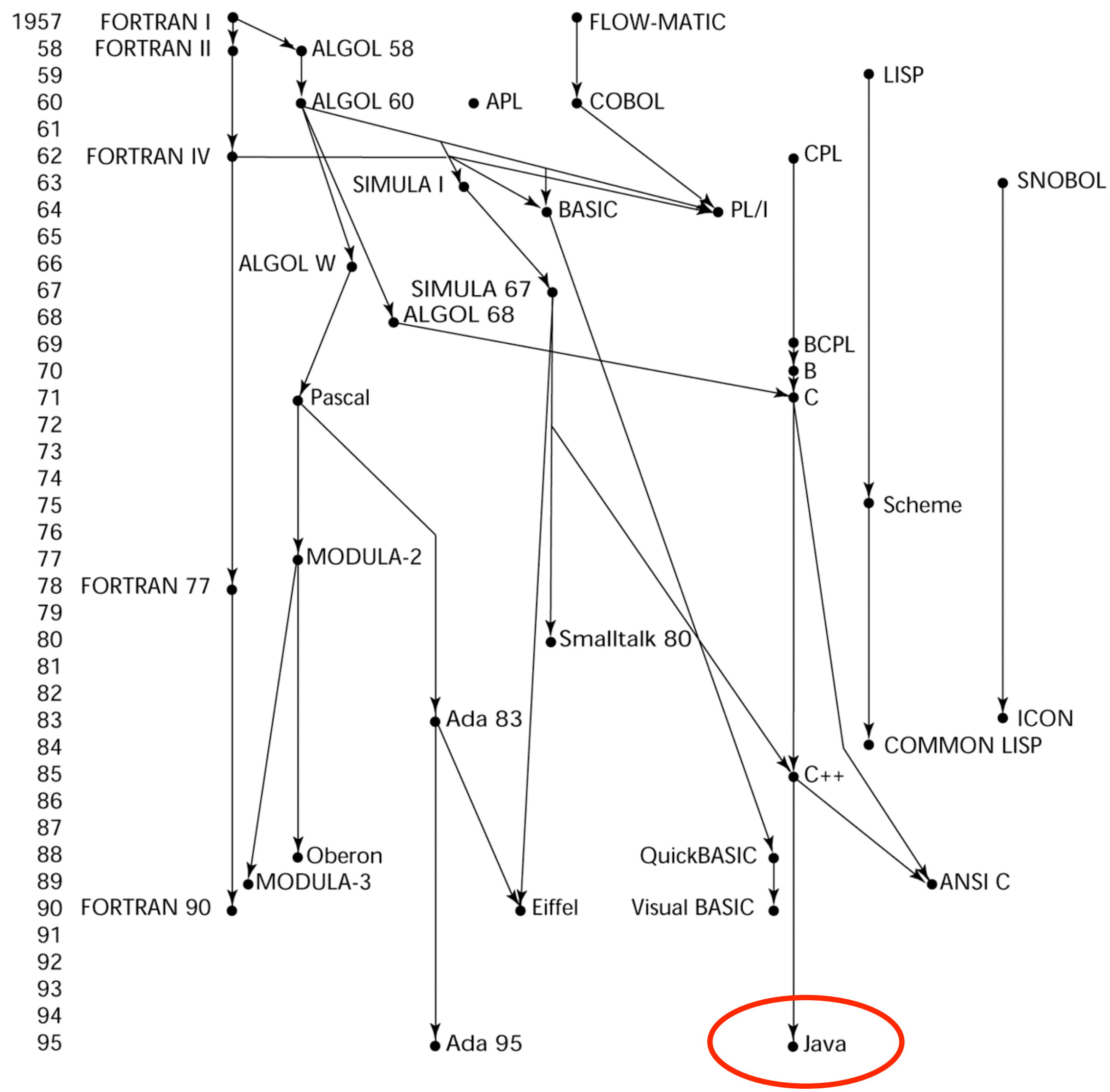
C++ (1985)

- Developed at Bell Labs by Bjarne Stroustrup
- Evolved from C and SIMULA 67
- Facilities for object-oriented programming, taken partially from SIMULA 67, were added to C
- Also has exception handling
- A large and complex language, in part because it supports both procedural and OO programming
- Rapidly grew in popularity, along with OOP
- ANSI standard approved in November, 1997

C++ Related Languages

- Eiffel - a related language that supports OOP
  - (Designed by Bertrand Meyer - 1992)
  - Not directly derived from any other language
  - Smaller and simpler than C++, but still has most of the power
- Delphi (Borland)
  - Pascal plus features to support OOP
  - More elegant and safer than C++

Genealogy of High-Level Languages



Java (1995)

- Developed at Sun in the early 1990s
- Based on C++
  - Significantly simplified (does not include **`struct, union, enum`**, pointer arithmetic, and half of the assignment coercions of C++)
  - Supports *only* OOP
  - No multiple inheritance
  - Has references, but not pointers
  - Includes support for applets and a form of concurrency
  - Portability was "Job #1"

Scripting Languages for the Web

- JavaScript
  - Used in Web programming (client-side) to create dynamic HTML documents
  - Related to Java only through similar syntax
- PHP
  - Used for Web applications (server-side); produces HTML code as output
- Perl
- JSP
- Python

C#

- Part of the .NET development platform
- Based on C++ and Java
- Provides a language for component-based software development
- All .NET languages (C#, Visual BASIC.NET, Managed C++, J#.NET, and Jscript.NET) use Common Type System (CTS), which provides a common class library
- Likely to become widely used

Some Important Special Purpose Languages

- SQL
  - Relational Databases
- LaTeX
  - Document processing and typesetting
- HTML
  - Web page
- XML
  - Platform independent data representation
- UML
  - Software system specification
- VHDL
  - Hardware description language

Website with lots of examples in different programming languages old and new

http://www.ntecs.de/old-hp/uu9r/lang/html/lang.en.html#_link_sather

Strongly recommended for the curious mind!

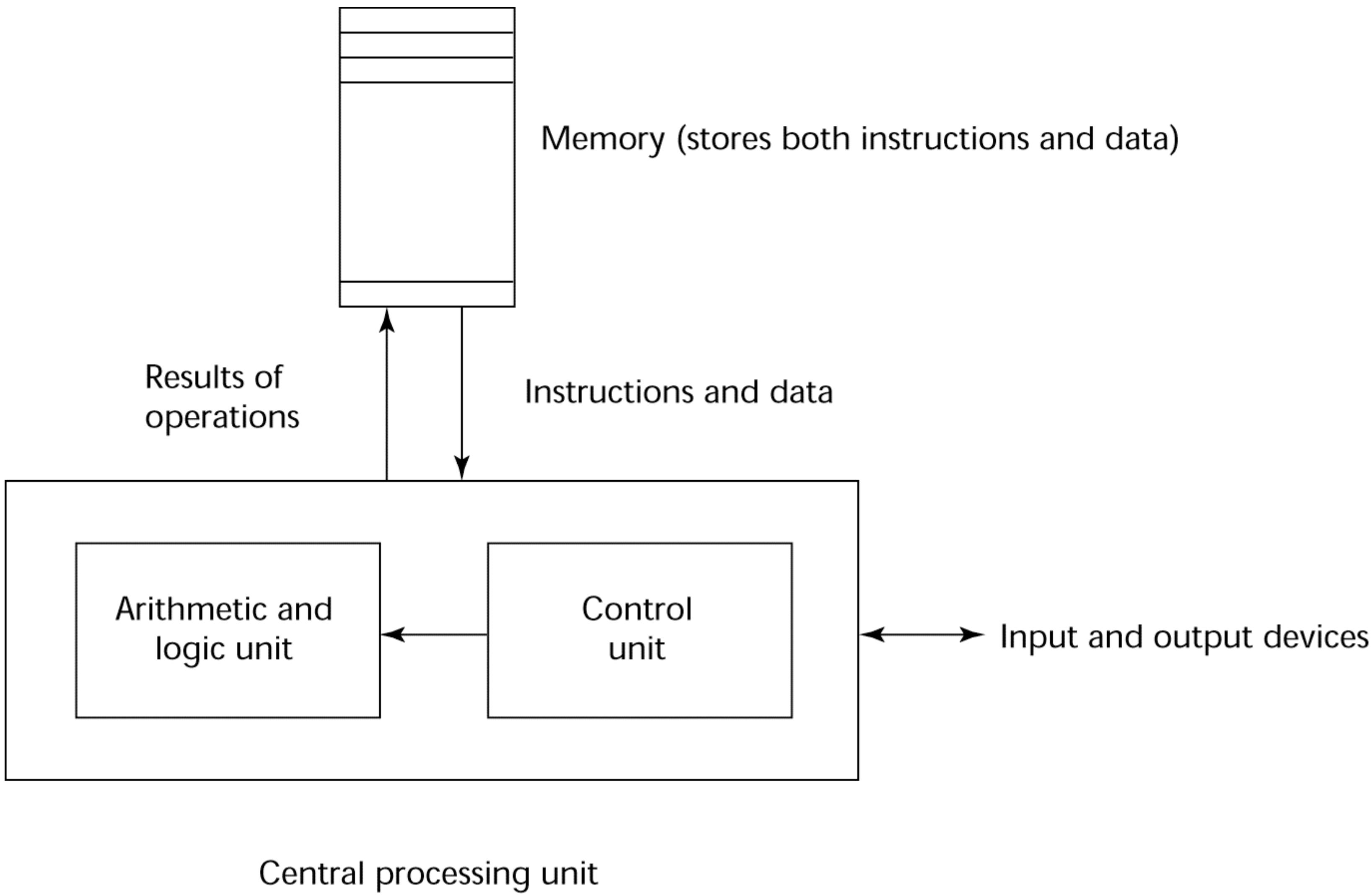
END OF LECTURE 1

EXTRA SLIDES

Influences on Language Design

- Computer architecture: Von Neumann
- We use imperative languages, at least in part, because we use von Neumann machines
  - Data and programs stored in same memory
  - Memory is separate from CPU
  - Instructions and data are piped from memory to CPU
- Basis for imperative languages
  - Variables model memory cells
  - Assignment statements model piping
  - Iteration is efficient

Von Neumann Architecture



LISP

- Pioneered functional programming
  - No need for variables or assignment
  - Control via recursion and conditional expressions
- Still the dominant language for AI
- COMMON LISP and Scheme are contemporary dialects of LISP
- ML, Miranda, and Haskell are related languages

Zuse's Plankalkül - 1945

- Never implemented
- Advanced data structures
  - floating point, arrays, records
- Invariants

Plankalkül

- Notation:

  A[7] = 5 * B[6]

```
    |  5  *  B  =>  A
V   |  6             7        (subscripts)
S   |  1.n           1.n      (data types)
```

Pseudocodes - 1949

- What was wrong with using machine code?
  - Poor readability
  - Poor modifiability
  - Expression coding was tedious
  - Machine deficiencies--no indexing or floating point

Pseudocodes

- Short code; 1949; BINAC; Mauchly
  - Expressions were coded, left to right
  - Some operations:

    1n => (n+2)nd power

    2n => (n+2)nd root

    07 => addition

Pseudocodes

- Speedcoding; 1954; IBM 701, Backus
  - Pseudo ops for arithmetic and math functions
  - Conditional and unconditional branching
  - Autoincrement registers for array access
  - Slow!
  - Only 700 words left for user program

Pseudocodes

- Laning and Zierler System - 1953
  - Implemented on the MIT Whirlwind computer
  - First "algebraic" compiler system
  - Subscripted variables, function calls, expression translation
  - Never ported to any other machine

ALGOL 58

- Comments:
  - Not meant to be implemented, but variations of it were (MAD, JOVIAL)
  - Although IBM was initially enthusiastic, all support was dropped by mid-1959

COBOL - 1960

- Sate of affairs
  - UNIVAC was beginning to use FLOW-MATIC
  - USAF was beginning to use AIMACO
  - IBM was developing COMTRAN

COBOL

- Based on FLOW-MATIC
- FLOW-MATIC features:
  - Names up to 12 characters, with embedded hyphens
  - English names for arithmetic operators (no arithmetic expressions)
  - Data and code were completely separate
  - Verbs were first word in every statement

COBOL

- First Design Meeting (Pentagon) - May 1959
- Design goals:
  - Must look like simple English
  - Must be easy to use, even if that means it will be less powerful
  - Must broaden the base of computer users
  - Must not be biased by current compiler problems
- Design committee members were all from computer manufacturers and DoD branches
- Design Problems: arithmetic expressions? subscripts? Fights among manufacturers

Ada 95

- Ada 95 (began in 1988)
  - Support for OOP through type derivation
  - Better control mechanisms for shared data (new concurrency features)
  - More flexible libraries