CENG204 - Programming Languages Concepts

Asst. Prof. Dr. Emre ŞATIR

Lecture 4
Evolution of the Major Programming Languages (Part 1)

Lecture 4 Topics

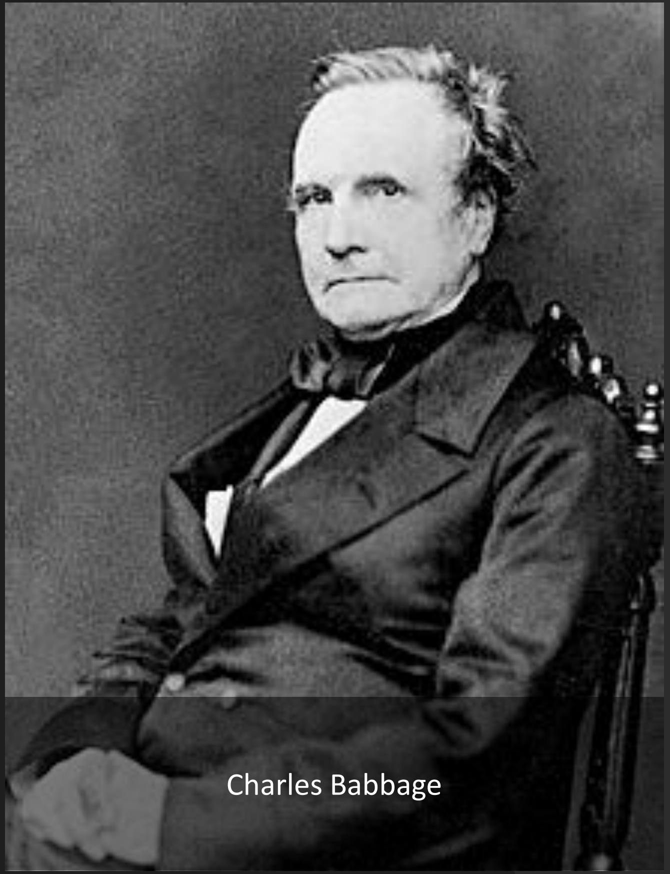
- First Ideas
- Machine and Assembly Languages
- The IBM 704 and Fortran
- Functional Programming: Lisp
- The First Step Toward Sophistication: ALGOL 60
- Computerizing Business Records: COBOL
- The Beginnings of Timesharing: Basic
- Everything for Everybody: PL/I
- Two Early Dynamic Languages: APL and SNOBOL

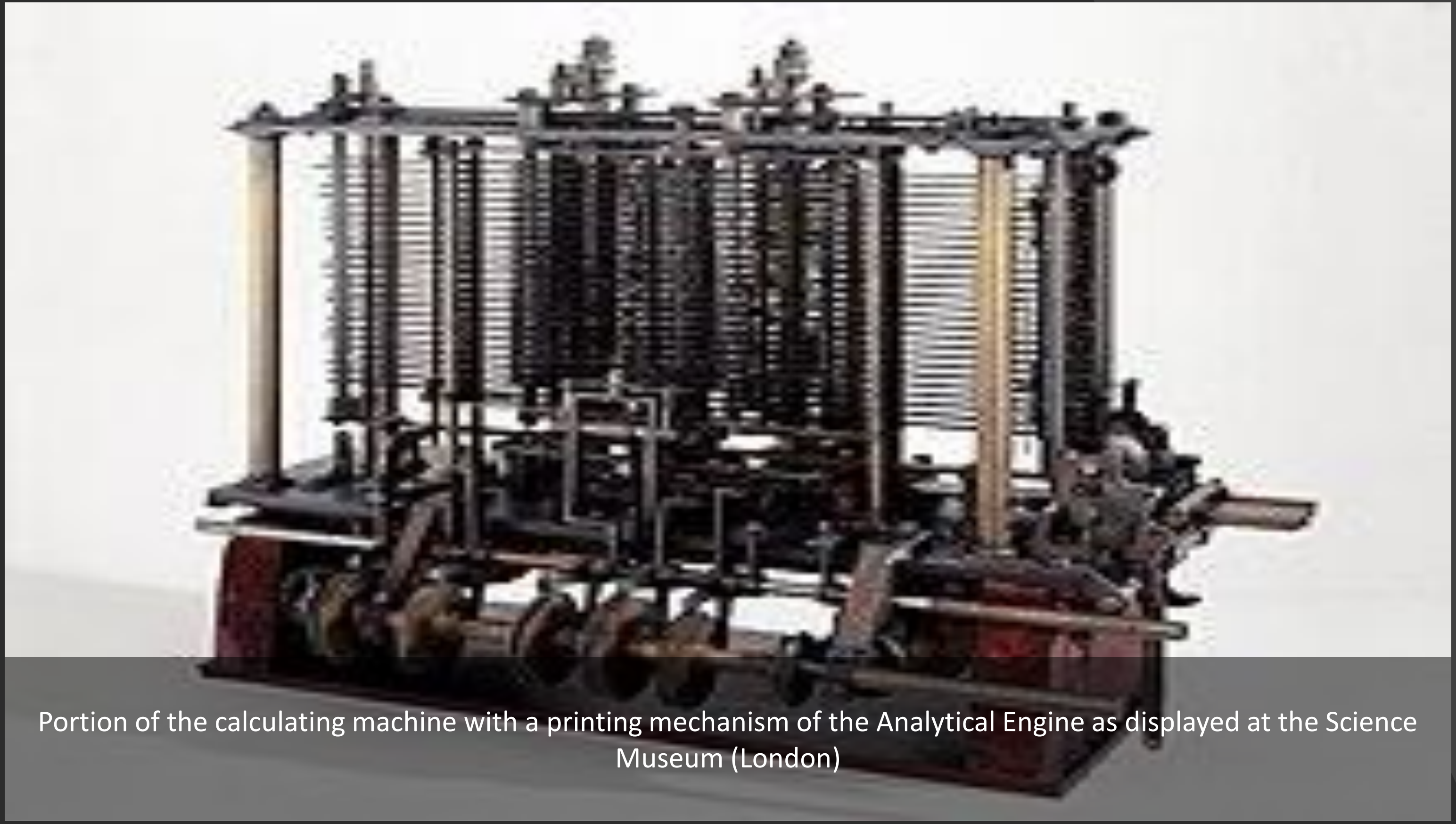
First Ideas

First Ideas

- The dates were in the 1800s when the famous English mathematician **Charles Babbage** introduced the idea of the “**Analytical Engine**”, which is considered the ancestor of today's modern computers.
- The Analytical Engine was essentially a “programmable computer idea”.
- The Analytical Engine was a proposed mechanical general-purpose computer designed by Charles Babbage. It was first described in 1837 as the successor to Babbage’s “**Difference Engine**”, (in 1822) which was a design for a simpler mechanical calculator.

First Ideas


Charles Babbage


Portion of the calculating machine with a printing mechanism of the Analytical Engine as displayed at the Science Museum (London)

First Ideas

- The first person considered to be a "programmer" is a woman, and she was Ada Lovelace (Byron) (1815-1852), daughter of the famous English poet Lord Byron.
- While translating an article about the “Analytical Engine” from French to English, Loveless also published her own ideas and demonstrated algorithmically that this machine could be programmed.
- Loveless presented an idea of how Bernoulli Numbers can be computed algorithmically on this machine.
- Ada Lovelace died at a very young age of 37. The “Ada Programming Language”, developed for the US Department of Defense in the 1970s, was named after her for her contributions to the field.

Ada Lovelace:The Mother of Programming

- Her notes on the Analytical Engine included what is now recognized as the first algorithm designed for implementation on a computer.
- Lovelace also envisioned the potential of computers beyond mere calculation, foreseeing their applications in music and art.

First Ideas

- English mathematician and cryptologist Alan Turing, who lived between 1912 and 1954, is considered the **founder** of “Computer Science”.
- With the machine he designed during the Second World War, he broke the Germans' Enigma codes and changed the course of the war.
- Historians have stated that thanks to this, the war was shortened considerably, and millions of lives were saved.

First Ideas

- Turing's Turing Machine was <u>not</u> a mechanical or electronic device but rather a "mathematical model" designed as a theoretical concept for computation. It was <u>not</u> a physical machine but an abstract computational model.
- However, Turing was part of the team that developed **Colossus**, the first programmable electronic computer. Colossus (1943-1944) was built to decrypt Nazi Enigma codes during World War II and operated using electronic circuits.
- On the other hand, **ENIAC** is recognized as the first "general-purpose electronic computer". Unlike Colossus, it was designed for <u>general computations</u>.

First Ideas

- ENIAC (Electronic Numerical Integrator And Computer) is the first electrically powered general-purpose computer capable of electronic data processing (1946).
- Built by American scientists (John Mauchly and J. Presper Eckert) during World War II, ENIAC fit on an area of approximately 167 square meters and weighed 30 tons.

Machine and Assembly Languages

Zuse’s Plankalkül

- The first programming language in history, designed by German Scientist Konrad Zuse in 1943, finished in 1945 but not released until 1972 (designed in 1945, but not published until 1972).
- Plankalkül means “Program Calculus”.
- Never implemented (remained in theory, not put into practice).
- The programming language whose first compiler was completed by the Technical University of Berlin in 2000, 5 years after Konrad Zuse's death, is claimed by Zuse to be the first high-level language.
- Includes advanced data structures
  - floating point, arrays, records

The first high level programming language,
Plankalkül
implemented on an electromechanical computer.
Inventor
Konrad Zuse
1942-1945

pseudocodes

Pseudocodes

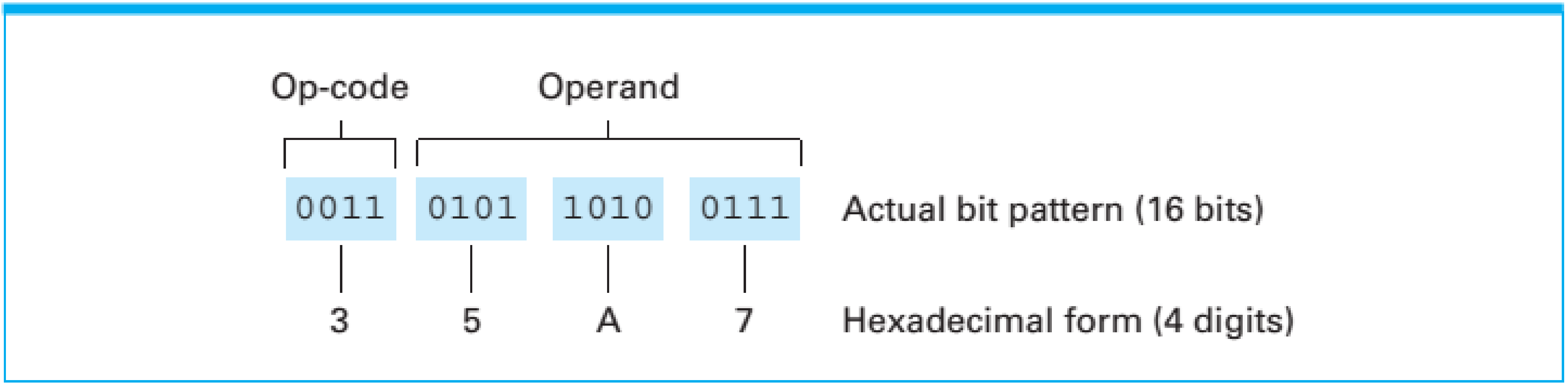
- Note that the word pseudocode is used here in a <u>different</u> sense than its <u>contemporary meaning</u>. We call the languages discussed in this section pseudocodes because <u>that’s what they were named</u> at the time they were developed and used (the late 1940s and early 1950s). However, they are clearly <u>not</u> pseudocodes in the contemporary sense.
- The computers that became available in the late 1940s and early 1950s were far <u>less usable</u> than those of today. In addition to being <u>slow</u>, <u>unreliable</u>, <u>expensive</u>, and having extremely <u>small memories</u>, the machines of that time were <u>difficult to program.</u>

Machine Languages

- There were no high-level programming languages or even assembly languages, so programming was done in machine code.
- Machine language is a language that a computer can directly understand and depends on the hardware design of the computer.
- Machine language is the commands given to the hardware parts of the computer.
- Code that runs on one computer will not run on another computer using a different CPU or memory design (Machine language is not portable, it is machine dependent).

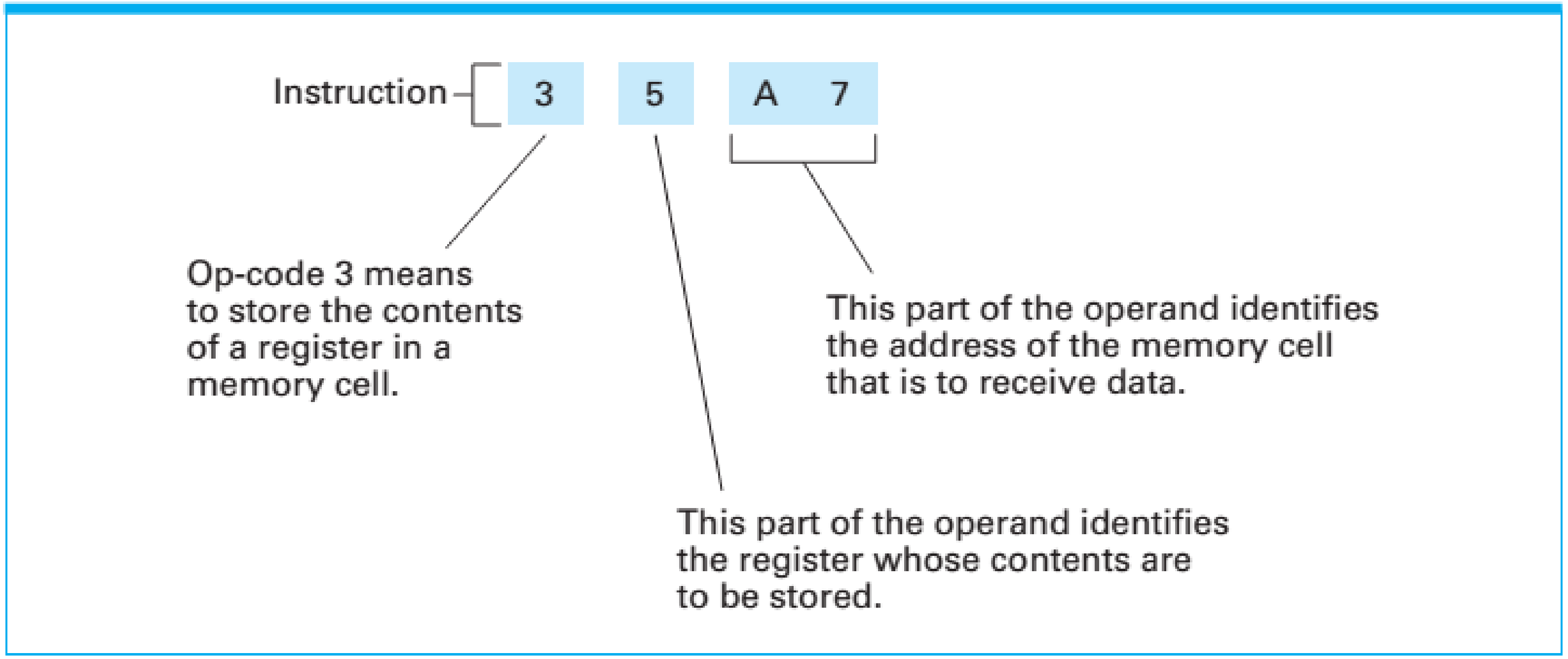
An Example Machine Language

The composition of an **instruction** for an example machine language.



An Example Machine Language

The instruction 35A7 (hexadecimal) translates to the statement "STORE the bit pattern found in register 5 in the memory cell whose address is A7.

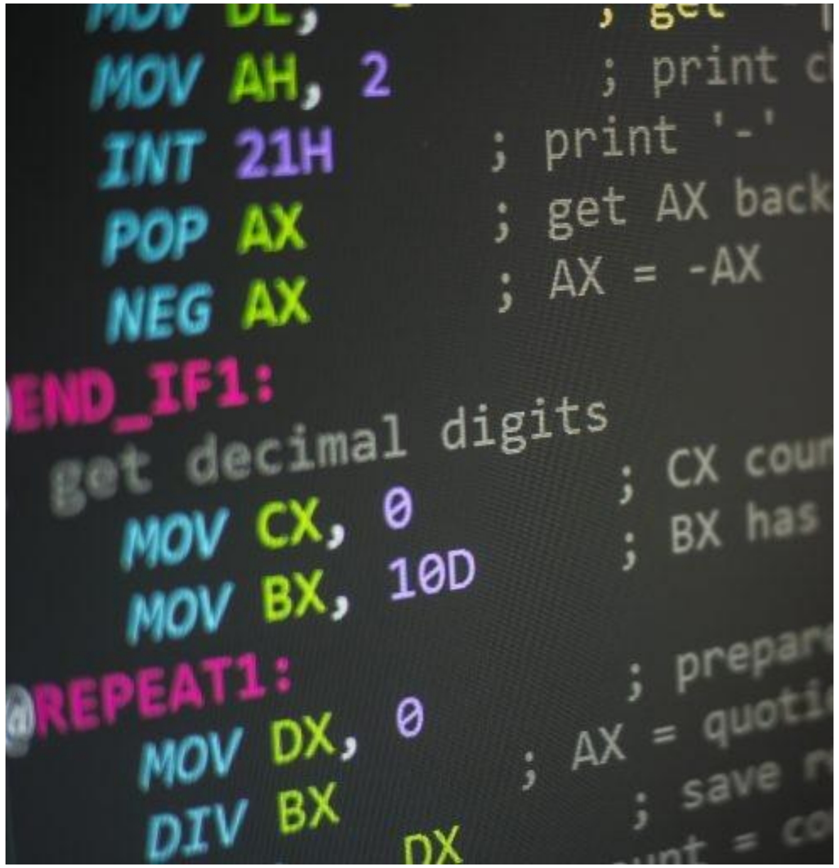

Instruction
3
5
A 7
Op-code 3 means to store the contents of a register in a memory cell.
This part of the operand identifies the address of the memory cell that is to receive data.
This part of the operand identifies the register whose contents are to be stored.

Machine Languages

- What was wrong with using machine code?
  - Poor readability
  - Poor modifiability
  - Coding was tedious
  - Error prone
- These are standard problems with all machine languages and were the primary motivations for inventing assembly languages.

Assembly Languages

- In order to partially remove the difficulty of understanding machine language, “**symbolic machine languages**”, namely assembly languages (sometimes referred as assemblers), were developed.
- Unlike machine languages, assembly languages consist of some <u>abbreviations in English</u> instead of 0 and 1.
- Researchers simplified the programming process by developing “notational systems” by which instructions could be represented in “mnemonic” rather than “numeric” form.



```
    MOV AH, 2          ; print c
    INT 21H          ; print '-'
    POP AX           ; get AX back
    NEG AX           ; AX = -AX
END_IF1:
get decimal digits
    MOV CX, 0              ; CX coun
    MOV BX, 10D            ; BX has
@REPEAT1:                  ; prepar
    MOV DX, 0          ; AX = quoti
    DIV BX                 ; save r
```

Assembly Languages

- For example, the instruction

  “move the contents of register 5 to register 6”
- would be expressed as

```
4056
```

- using the machine language, whereas in a mnemonic system it might appear as

```
MOV R5, R6
```

Assembly Languages

- As a more extensive, the machine language routine

```
156C
166D
5056
306E
C000
```

- which adds the contents of memory cells 6C and 6D and stores the result at memory location 6E, might be expressed as

```
LD R5,Price
LD R6,ShippingCharge
ADDI R0,R5 R6
ST R0,TotalCost
HLT
```

- using mnemonics. (Here we have used LD, ADDI, ST, and HLT to represent “load”, “add”, “store”, and “halt”. Moreover, we have used the descriptive names (called as **identifiers**) Price, ShippingCharge, and TotalCost to refer to the memory cells at locations 6C, 6D, and 6E, respectively.

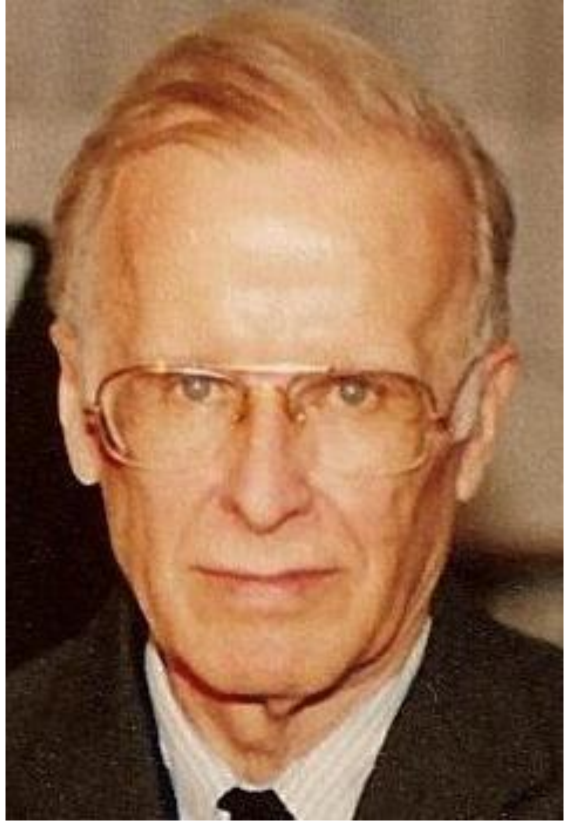
Assembly Languages

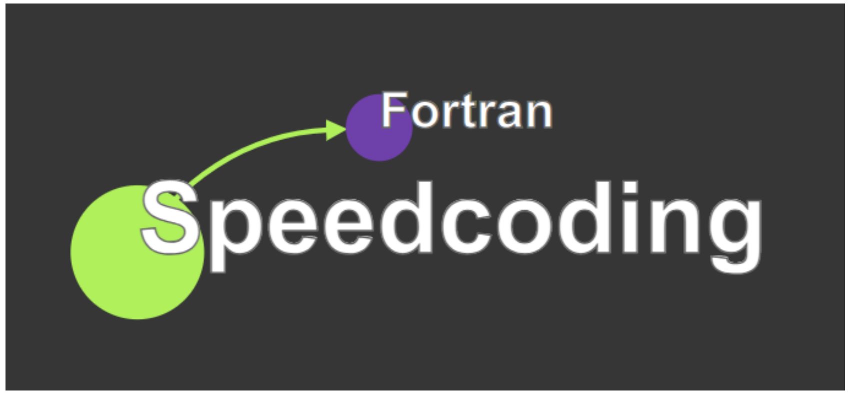
- Although assembly languages have many advantages over their machine language counterparts, they still fall short of providing the ultimate programming environment.
- After all, the primitives used in an assembly language are essentially the same as those found in the corresponding machine language. The difference is simply in the syntax used to represent them.
- Thus, a program written in an assembly language is inherently machine dependent—that is, the instructions within the program are expressed in terms of a particular machine's attributes.
- In turn, a program written in assembly language cannot be transported (i.e not portable) to another computer design because it must be rewritten to conform to the new computer's register configuration and instruction set.

```
01 -    06 abs value     1n (n+2)nd power
02 )    07 +             2n (n+2)nd root
03 =    08 pause         4n if <= n
04 /    09 (             58 print and tab
```

Short Code

- Short Code, was developed by John Mauchly in 1949 for the BINAC computer, which was one of the first successful stored-program electronic computers.
- Short Code was not translated to machine code; rather, it was implemented with a pure interpreter. At the time, this process was called **automatic programming**.
- It clearly simplified the programming process (according to machine languages), but at the expense of execution time. Short Code interpretation was approximately 50 times slower than machine code.



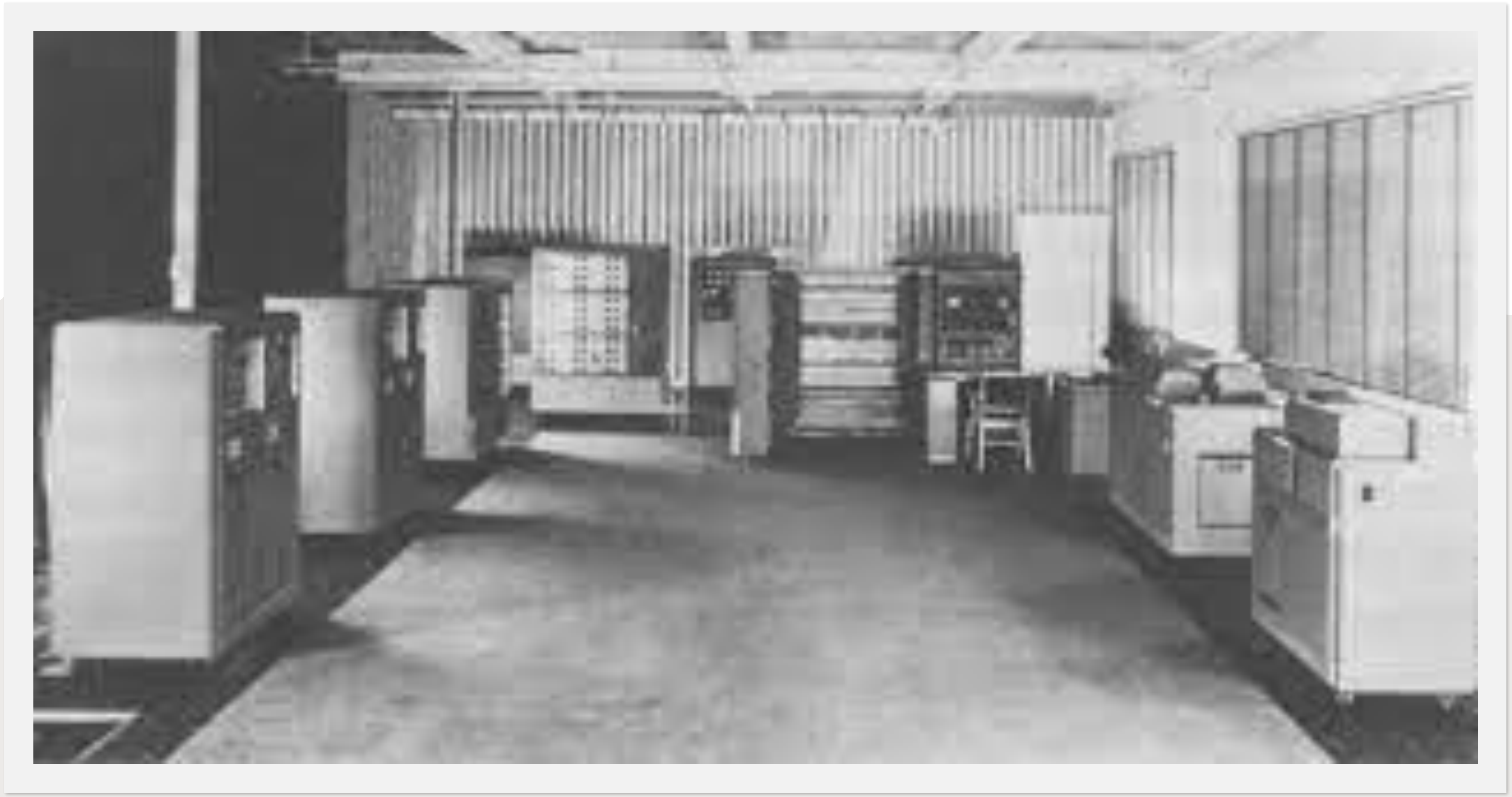


Speedcoding

- In other places, interpretive systems were being developed that extended machine languages to include floating-point operations.
- The **Speedcoding System** developed by John Backus for the IBM 701 is an example of such a system.
- The system included pseudo instructions for the four arithmetic operations on floating-point data, as well as operations such as "square root", "sine", "arc tangent", "exponent", and "logarithm".
- To get an idea of the limitations of such systems, consider that the remaining usable memory after loading the interpreter was only 700 words and that the add instruction took 4.2 milliseconds to execute (too slow!).
- Backus claimed that problems that could take two weeks to program in machine code could be programmed in a few hours using Speedcoding.

Speedcoding

- The IBM 701 did not have hardware support for floating-point arithmetic. Instead, the Speedcoding system, developed by John Backus, provided a software-based solution for floating-point operations.
- Speedcoding was an interpreted system that enabled **floating-point calculations** and **array indexing** “through software” rather than hardware.
- While Speedcoding simplified programming, it came with a significant performance cost, running 10 to 50 times slower than machine code.



THE IBM 701

Programming So Far…

- Slow running (Short Code and Speedcoding)
- Poor reliability
- High cost
- Limited memory
- Problem of “absolute addressing”
- Array indexing not supported
- Floating point arithmetic not supported (lack of hardware)

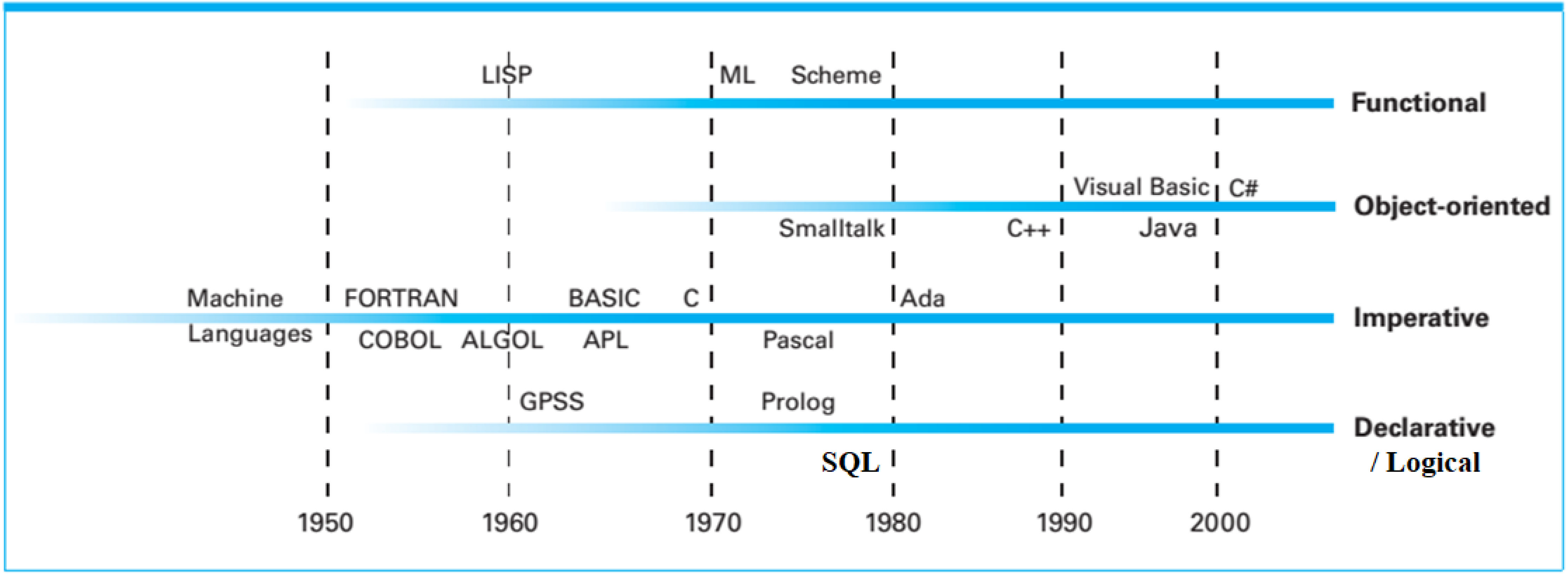
Programming So Far…

- Between 1946 and 1950, the concept of operating system did not exist yet, and therefore the electronic structure of the computer (hardware) had to be known very well.
- The first thing that comes to mind when programming is mentioned was to design a circuit for the problem to be solved.
- Therefore, only experts could program, and computers were not widely used.

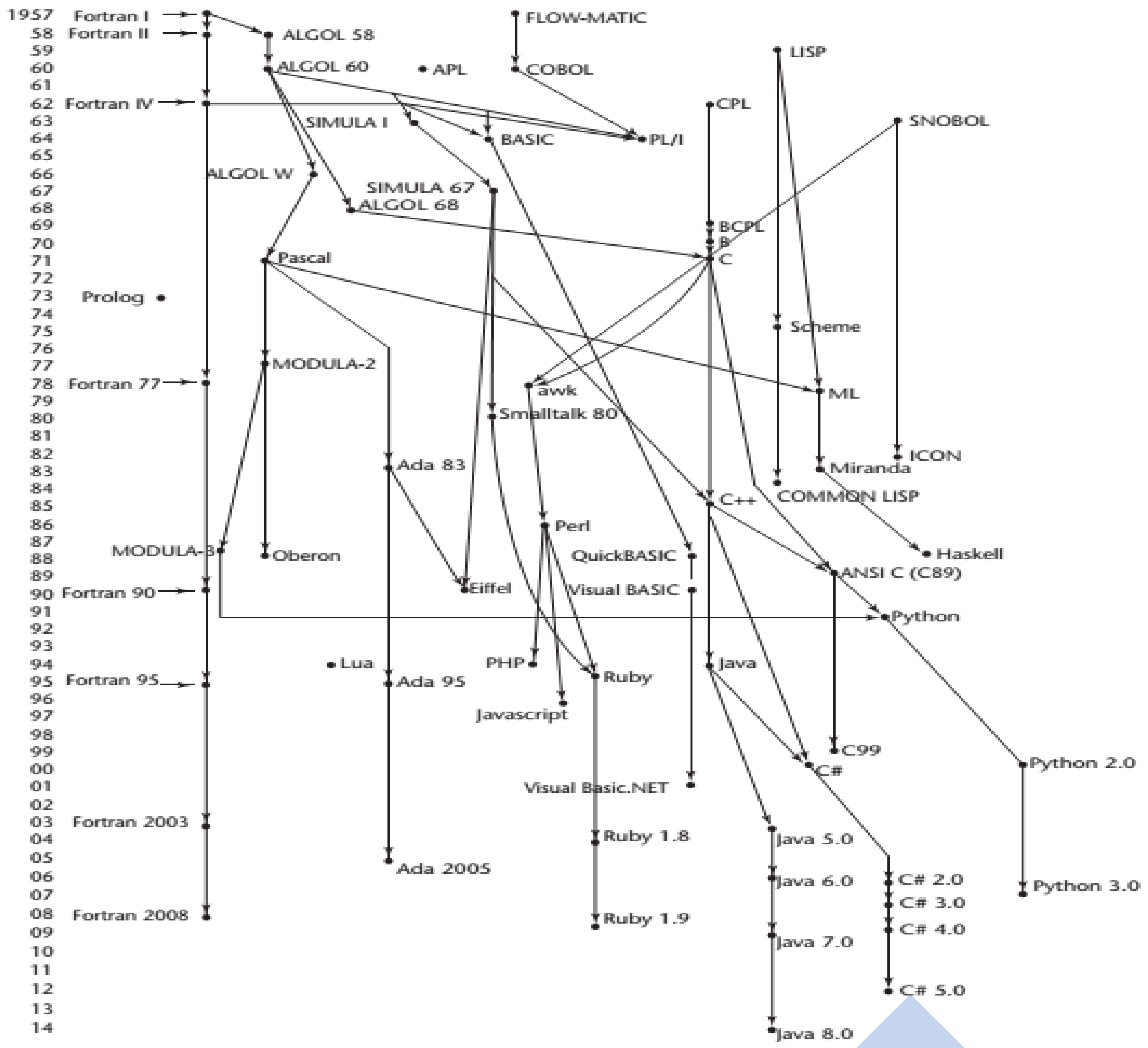
Programming So Far…

- It was thought that it was necessary to create a computer language consisting of statements similar to the words used in everyday language.
- This language had to be easy to learn, understandable and at the same time suitable for mathematical applications.
- The first language developed for this purpose was **Fortran**.

The evolution of programming paradigms



Evolution of the Major Programming Languages

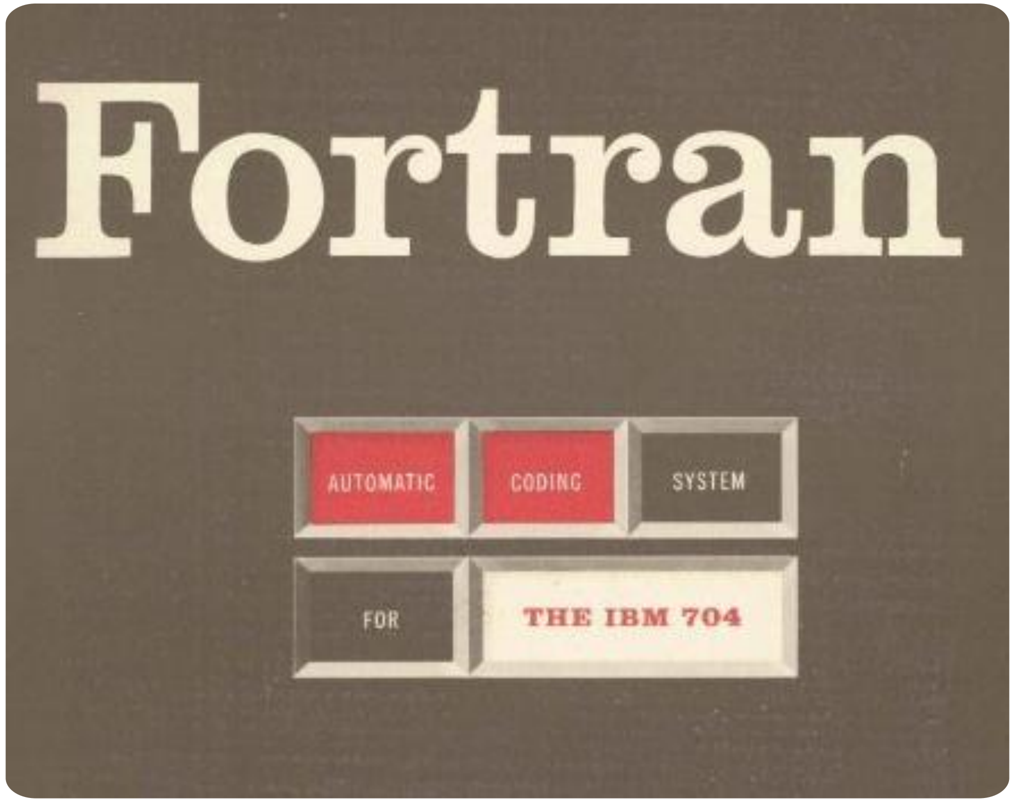


The IBM 704 and Fortran

Compiler Idea

- The "compiler" came up with the idea of **Grace Hopper** (The UNIVAC Compiling System team leader).
- She was an American computer scientist, mathematician, and United States Navy rear admiral.
- Hopper was the first to devise the theory of machine-independent programming languages, and the FLOW-MATIC programming language she created using this theory was later extended to create COBOL, an early high-level programming language still in use today.
- According to this idea, instead of translating the code into machine language one by one during each run (systems like Short Code or Speedcoding), it will be translated once and then the program can be run repeatedly.
- Thus, the codes written in a higher-level language gained the opportunity to work at their original speed.

The IBM 704 and Fortran



- Certainly, one of the greatest single advances in computing came with the introduction of the IBM 704 in 1954, in large measure because its capabilities prompted the development of Fortran (**FOR**mula **TRAN**slating).
- Fortran was developed by John Backus at IBM in 1954.
- New IBM 704 had index registers (for stepping through arrays) and floating point hardware.

The IBM 704 and Fortran

- In 2 years, John Backus produced the Fortran compiler, consisting of 51.000 lines of machine code and stored on a tape unit.
- Fortran is literally the first high-level programming language to have a compiler (The Fortran language is considered the first of the high-level languages).
- Fortran was quite easy to understand compared to machine language, and the language had good features for mathematical applications.
- Fortran became commercially available in 1957.
- John Backus developed the "Backus Naur Form (BNF)" in 1959, which has become a standard notation for describing the syntax rules of a high-level language.

The IBM 704 and Fortran

- Fortran contained "do", input-output and assignment statements.
- Fortran facilitated matrix operations, systems of equations, and differential equations.
- Example “if” structure
  - `if (arithmetic expression) n1, n2, n3`
- Example “do” structure
  - `do variable = first_value, last_value`
- Fortran Versions: Fortran 0 (1954 - not implemented), Fortran I (1957), Fortran II (1958), Fortran IV (Fortran 66), Fortran 77, Fortran 90, Fortran 95, Fortran 2003, Fortran 2008.

Fortran Evaluation

- One of the features of Fortran I, and all of its successors before 90, that allows highly optimizing compilers was that the types and storage for all variables are fixed before run time. No new variables or space could be allocated during execution.
- It dramatically changed the way computers are used. This is, of course, in large part due to its being the first widely used high-level language.
- Alan Perlis, one of the designers of ALGOL 60, said of Fortran in 1978, “Fortran is the *lingua franca* of the computing world. … it has survived and will survive because it has turned out to be a remarkably useful part of a very vital commerce”

An Example of a Fortran 95 Program

```
! Fortran 95 Example program
!  Input:   An integer, List_Len, where List_Len is less
!           than 100, followed by List_Len-Integer values
!  Output: The number of input values that are greater
!           than the average of all input values
Implicit none
Integer Dimension(99) :: Int_List
Integer :: List_Len, Counter, Sum, Average, Result
Result= 0
Sum = 0
Read *, List_Len
If ((List_Len > 0) .AND. (List_Len < 100)) Then
! Read input data into an array and compute its sum
   Do Counter = 1, List_Len
       Read *, Int_List(Counter)
       Sum = Sum + Int_List(Counter)
   End Do
```

An Example of a Fortran 95 Program (continued)

```
! Compute the average
   Average = Sum / List_Len
! Count the values that are greater than the average
   Do Counter = 1, List_Len
      If (Int_List(Counter) > Average) Then
         Result = Result + 1
      End If
   End Do
! Print the result
   Print *, 'Number of values > Average is:', Result
Else
   Print *, 'Error - list length value is not legal'
End If
End Program Example
```

Functional Programming: Lisp

Functional Programming: Lisp

- The first functional programming language was invented to provide language features for list processing, the need for which grew out of the first applications in the area of artificial intelligence (AI).
- LISP stands for "**LIS**t **P**rocessing Language" and is designed at MIT by John McCarthy (1958).
- AI research needed a language to
  - Process data in lists (rather than arrays)
  - Symbolic computation (rather than numeric)
- Only two data types: atoms and lists.

Processes in Functional Programming

- Lisp was designed as a functional programming language.
- All computation in a purely functional program is accomplished by applying functions to arguments.
- Neither the assignment statements nor the variables that abound in imperative language programs are necessary in functional language programs.
- Furthermore, repetitive processes can be specified with recursive function calls, making iteration (loops) unnecessary.
- These basic concepts of functional programming make it significantly different from programming in an imperative language.

The Syntax of Lisp

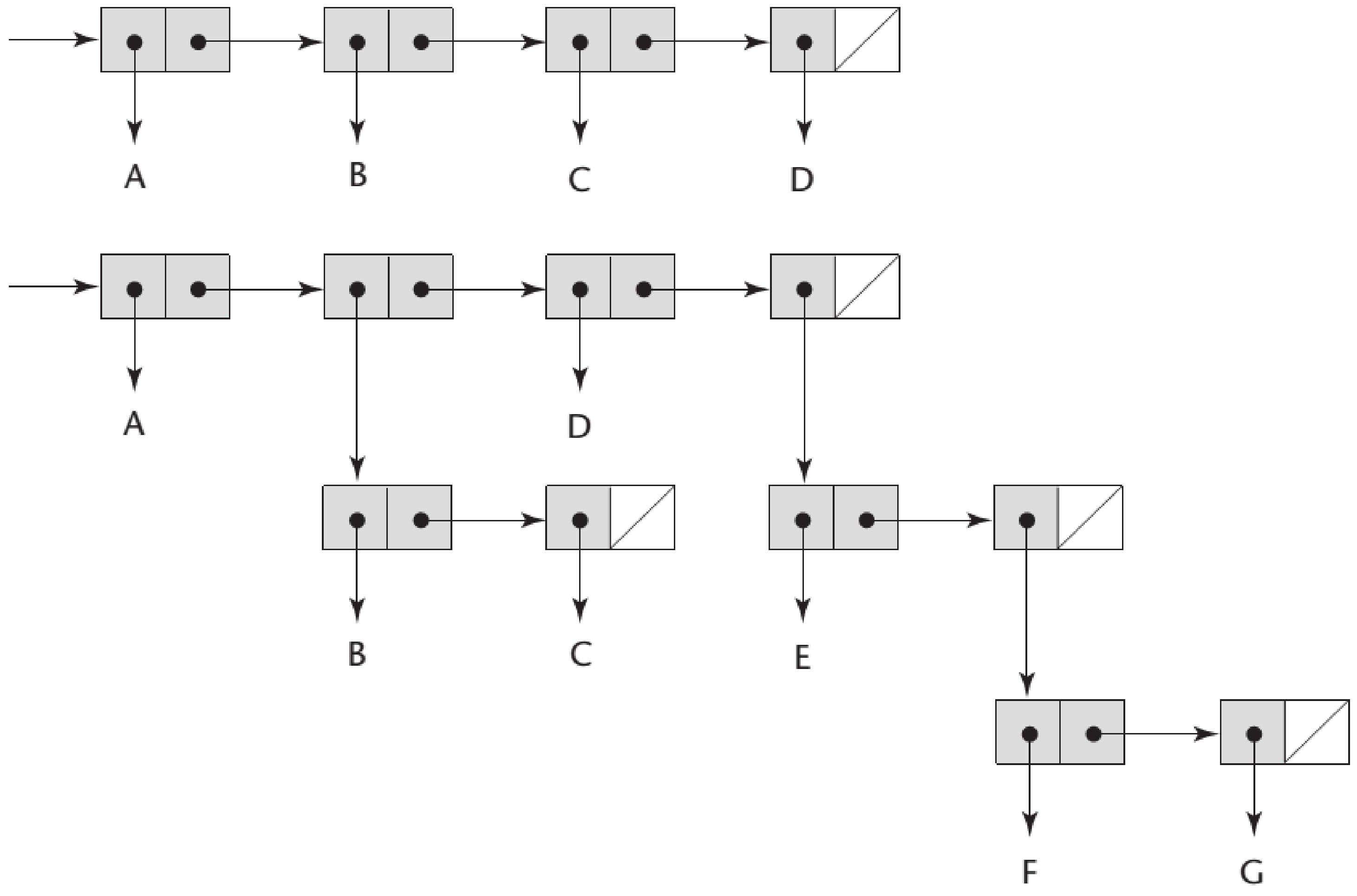
- Lisp is very <u>different</u> from the imperative languages, both because it is a functional programming language and because the <u>appearance of Lisp programs is so different</u> from those in languages like Java or C.
- For example, the syntax of <u>Java</u> is a complicated <u>mixture of English and algebra</u>, while Lisp's syntax is a <u>model of simplicity</u>.
- In Lisp, <u>program code</u> and <u>data</u> have exactly the same form: **parenthesized lists**. For example, consider the list below:

  ```
  (A B C D)
  ```

- When interpreted as **data**, *it is a list of four elements*. When viewed as **code**, *it is the application of the function named A to the three parameters B, C, and D*.

```
(list '
(str a b c d)
(append list-a '(
(list? '(a b c))
b c d]
```

Internal Representation of Two Lisp Lists



An Example of a Lisp Program

```
; Lisp Example function
; The following code defines a Lisp predicate function
; that takes two lists as arguments and returns True
; if the two lists are equal, and NIL (false) otherwise
  (DEFUN equal_lists (lis1 lis2)
   (COND
     ((ATOM lis1) (EQ lis1 lis2))
     ((ATOM lis2) NIL)
     ((equal_lists (CAR lis1) (CAR lis2))
                (equal_lists (CDR lis1) (CDR lis2)))
     (T NIL)
   )
)
```

Lisp Evaluation

- Pioneered functional programming.
  - No need for variables or assignment
  - Control via recursion and conditional expressions
- ML (*MetaLanguage*), Miranda, Haskell, and F# (a .NET language) are also functional programming languages but use very different syntax.
- “Scheme” and “Common Lisp” are contemporary dialects of Lisp.

Scheme

λ

- Developed at MIT in mid 1970s.
- As a "small language" with simple syntax and semantics, Scheme is well suited to educational applications, such as courses in functional programming and general introductions to programming.

Common Lisp

- During the 1970s and early 1980s, a large number of different dialects of Lisp were developed and used.
- This led to the familiar problem of lack of portability among programs written in the various dialects.
- Common Lisp was created in an effort to rectify this situation.
- Common Lisp was designed by combining the features of several dialects of Lisp (including Scheme) developed in the early 1980s, into a single language.
- Being such an amalgam, Common Lisp is a relatively large and complex language.
- Its basis, however, is pure Lisp, so its syntax, primitive functions, and fundamental nature come from that language.

The First Step Toward Sophistication: ALGOL 60

ALGOL

- ALGOL stands for “**ALGO**rithmic **L**anguage”.
- In 1958 in Zurich, a commission of European and American researchers developed ALGOL 58, a high-level language inspired by Fortran I.
- ALGOL 60 strongly influenced subsequent programming languages and is therefore of central importance in any historical study of languages.
- Many principles developed in the Algol language are still used in modern programming languages.

ALGOL

- ALGOL 60 was the result of efforts to design a universal programming language for scientific applications.
- In many ways, ALGOL 58 was a descendant of Fortran.
- It generalized many of Fortran's features and added several new constructs and concepts.
- Some of the generalizations had to do with the goal of not tying the language to any particular machine, and others were attempts to make the language more flexible and powerful.
- A rare combination of simplicity and elegance emerged from the effort.

ALGOL 58

- Concept of type was formalized
- Names could be 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
- Assignment operator was :=
- **`if`** had an **`else-if`** clause
- No I/O - “would make it machine dependent”

ALGOL 60 Overview

- Modified ALGOL 58 at 6-day meeting in Paris.
- New features
  - Block structure (local scope).
  - Two parameter passing methods.
  - Subprogram recursion.
  - Stack-dynamic arrays.

  - Still <u>no I/O</u> and <u>no string handling.</u>

ALGOL 60 Evaluation

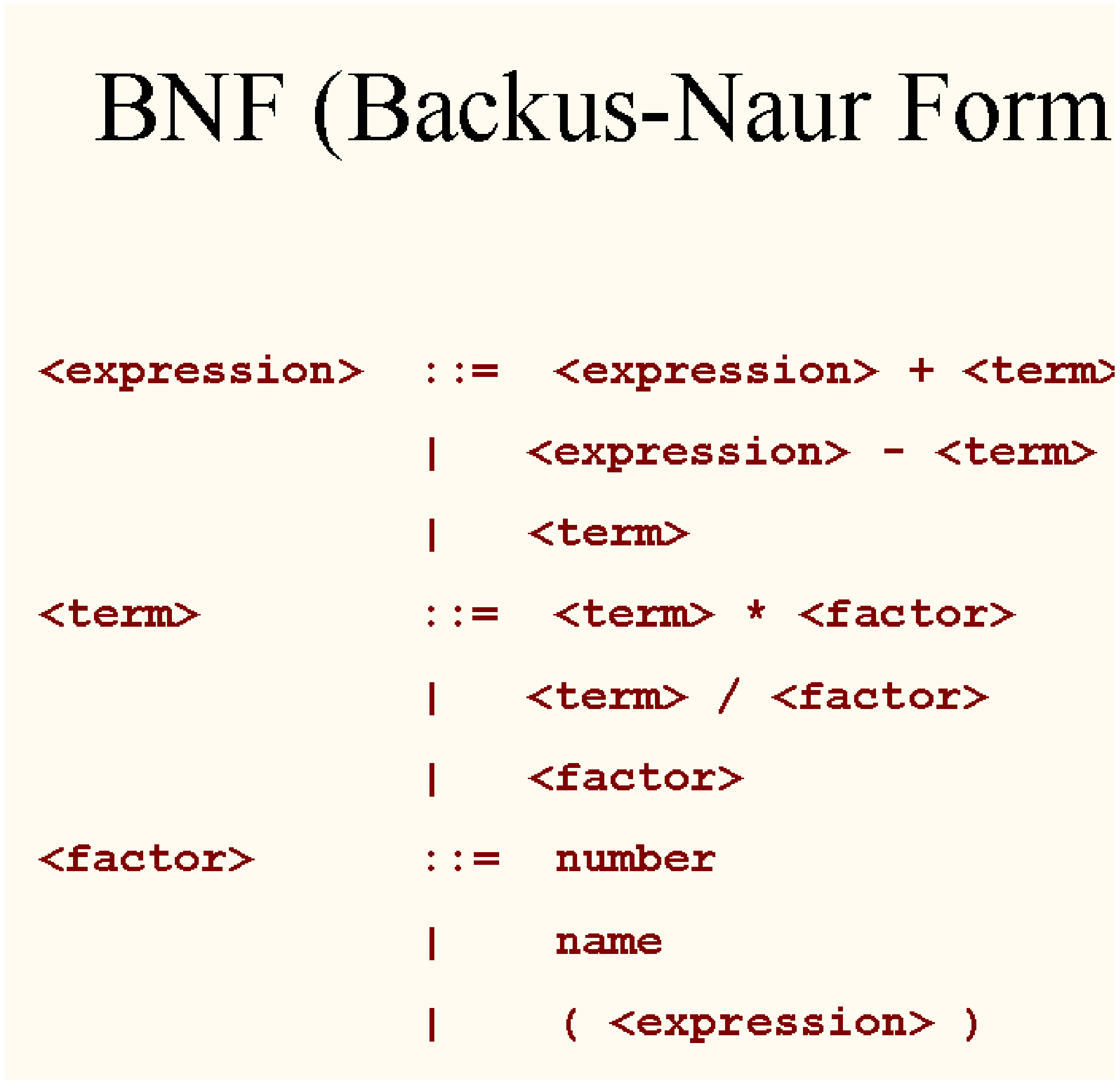
- Successes
  - It was the standard way to publish algorithms for over 20 years.
  - All subsequent imperative languages are based on it (direct or indirect descendants; examples include PL/I, SIMULA 67, ALGOL 68, C, Pascal, Ada, C++, Java, and C#)
  - It was the first time that an international group attempted to design a programming language.
  - It was the first language that was designed to be machine independent.
  - First language whose syntax was formally defined (BNF).

ALGOL 60 Evaluation (continued)

- Failure
  - Never widely used, especially in U.S.
  - Lack of support from IBM

ALGOL

- One of the most important contributions to computer science associated with ALGOL 60 is "**Backus Naur Form (BNF)**".
- It is the first language whose syntax was formally described.
- This successful use of the BNF formalism initiated several important fields of computer science: formal languages, parsing theory, and BNF-based compiler design.



BNF (Backus-Naur Form

```
<expression>  ::=  <expression> + <term>
                |   <expression> - <term>
                |   <term>
<term>        ::=  <term> * <factor>
                |   <term> / <factor>
                |   <factor>
<factor>      ::=  number
                |   name
                |   ( <expression> )
```

An Example of an ALGOL 60 Program

```
comment ALGOL 60 Example Program
 Input:  An integer, listlen, where listlen is less than
         100, followed by listlen-integer values
 Output: The number of input values that are greater than
         the average of all the input values ;
begin
  integer array intlist [1:99];
  integer listlen, counter, sum, average, result;

  sum := 0;
  result := 0;
  readint (listlen);
```

An Example of an ALGOL 60 Program (continued)

```
  if (listlen > 0) ∧ (listlen < 100)  then
     begin
comment Read input into an array and compute the average;
     for counter := 1 step 1 until listlen do
        begin
        readint (intlist[counter]);
        sum := sum + intlist[counter]
        end;
comment Compute the average;
     average := sum / listlen;
comment Count the input values that are > average;
     for counter := 1 step 1 until listlen do
        if intlist[counter] > average
          then result := result + 1;
comment Print result;
     printstring("The number of values > average is:");
     printint (result)
     end
  else
     printstring ("Error-input list length is not legal";
end
```

Computerizing Business Records: COBOL

COBOL

- COBOL stands for "**CO**mmon **B**usiness **O**riented **L**anguage".
- The story of COBOL is, in a sense, the opposite of that of ALGOL 60.
- Although it has been used for 65 years, COBOL has had little effect on the design of subsequent languages, except for PL/I.
- It may still be one of the most widely used languages in business applications, although it is difficult to be sure one way or the other.
- Perhaps the most important reason why COBOL has had little influence is that few have attempted to design a new language for business applications since it appeared. That is due in part to how well COBOL's capabilities meet the needs of its application area.
- Another reason is that a great deal of growth in business computing over the past 30 years has occurred in small businesses. In these businesses, very little software development has taken place. Instead, most of the software used is purchased as off-the-shelf packages for various general business applications.

FLOW-MATIC

- One compiled language for business applications, FLOW-MATIC, had been implemented in 1957 by Grace Hopper, but it belonged to one manufacturer, UNIVAC, and was designed for that company's computers.
- COBOL is based on FLOW-MATIC.

COBOL Design Process

- First Design Meeting (Pentagon) - May 1959
- Design goals
  - Must look like <u>simple English</u>
  - Must be <u>easy to use</u>, even if that means it will be less powerful
  - Must <u>broaden</u> the base of computer users
  - Must <u>not</u> be biased by current compiler problems
- Design committee members were all from computer manufacturers and DoD (Department of Defence) branches.

COBOL

- COBOL includes routines for printing reports, searching tables and sorting files that are very easy to use.
- Due to its structural features, it is easy to use with database management systems.
- Still widely used in business applications.

An Example Part of a COBOL Program

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PRODUCE-REORDER-LISTING.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. DEC-VAX.
OBJECT-COMPUTER. DEC-VAX.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT BAL-FWD-FILE ASSIGN TO READER.
    SELECT REORDER-LISTING ASSIGN TO LOCAL-PRINTER.

DATA DIVISION.
FILE SECTION.
FD  BAL-FWD-FILE
    LABEL RECORDS ARE STANDARD
    RECORD CONTAINS 80 CHARACTERS.

01  BAL-FWD-CARD.
    02 BAL-ITEM-NO           PICTURE IS 9(5).
    02 BAL-ITEM-DESC         PICTURE IS X(20).
    02 FILLER                PICTURE IS X(5).
    02 BAL-UNIT-PRICE        PICTURE IS 999V99.
    02 BAL-REORDER-POINT     PICTURE IS 9(5).
    02 BAL-ON-HAND           PICTURE IS 9(5).
    02 BAL-ON-ORDER          PICTURE IS 9(5).
    02 FILLER                PICTURE IS X(30).
FD  REORDER-LISTING
    LABEL RECORDS ARE STANDARD
    RECORD CONTAINS 132 CHARACTERS.

01  REORDER-LINE.
```

The Beginnings of Timesharing: Basic

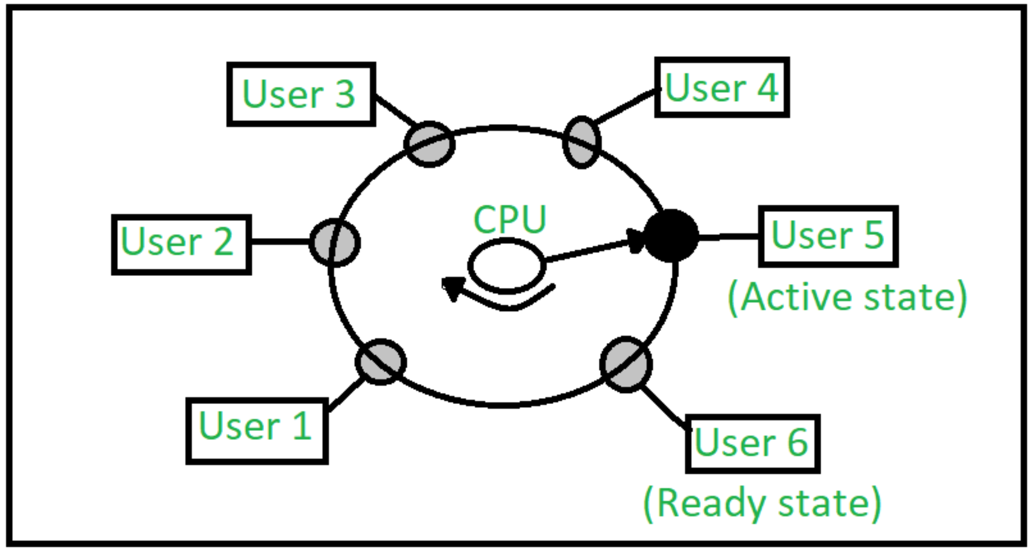
- Basic (Beginner’s All-purpose Symbolic Instruction Code) was originally designed at Dartmouth College by two mathematicians, John Kemeny and Thomas Kurtz, who, in the early 1960s, developed compilers for a variety of dialects of Fortran and ALGOL 60.
- Their science students generally had little trouble learning or using those languages in their studies. However, Dartmouth was primarily a liberal arts institution, where science and engineering students made up only about 25 percent of the student body.
- It was decided in the spring of 1963 to design a new language especially for liberal arts students.
- This new language would use terminals as the method of computer access.

Basic

- The goals of the system were as follows:
  - It must be easy for nonscience students to learn and use.
  - It must be “pleasant and friendly.”
  - It must provide fast turnaround for homework.
  - It must allow free and private access.
  - It must consider user time more important than computer time.
- Combination of these goals led to the timeshared aspect of Basic.
- Only with individual access through terminals by numerous simultaneous users could these goals be met in the early 1960s.

Basic

- Basic is the first programming language to establish a connection with a remote computer.
- First widely used language with time sharing.
- Current popular dialect: **Visual Basic**. It provides a visual programming environment (allows developers to design graphical user interfaces (GUIs) using a drag-and-drop approach) and supports object-oriented programming).



An Example of a Basic Program

```
REM  Basic Example Program
REM  Input:   An integer, listlen, where listlen is less
REM           than 100, followed by listlen-integer values
REM  Output: The number of input values that are greater
REM           than the average of all input values
  DIM intlist(99)
  result = 0
  sum = 0
  INPUT listlen
  IF listlen > 0 AND listlen < 100 THEN
REM  Read input into an array and compute the sum
    FOR counter = 1 TO listlen
      INPUT intlist(counter)
      sum = sum + intlist(counter)
    NEXT counter
REM  Compute the average
    average = sum / listlen
REM  Count the number of input values that are > average
    FOR counter = 1 TO listlen
      IF intlist(counter) > average
        THEN result = result + 1
    NEXT counter
REM  Print the result
    PRINT "The number of values that are > average is:";
           result
  ELSE
    PRINT "Error-input list length is not legal"
  END IF
END
```

Everything for Everybody: PL/I

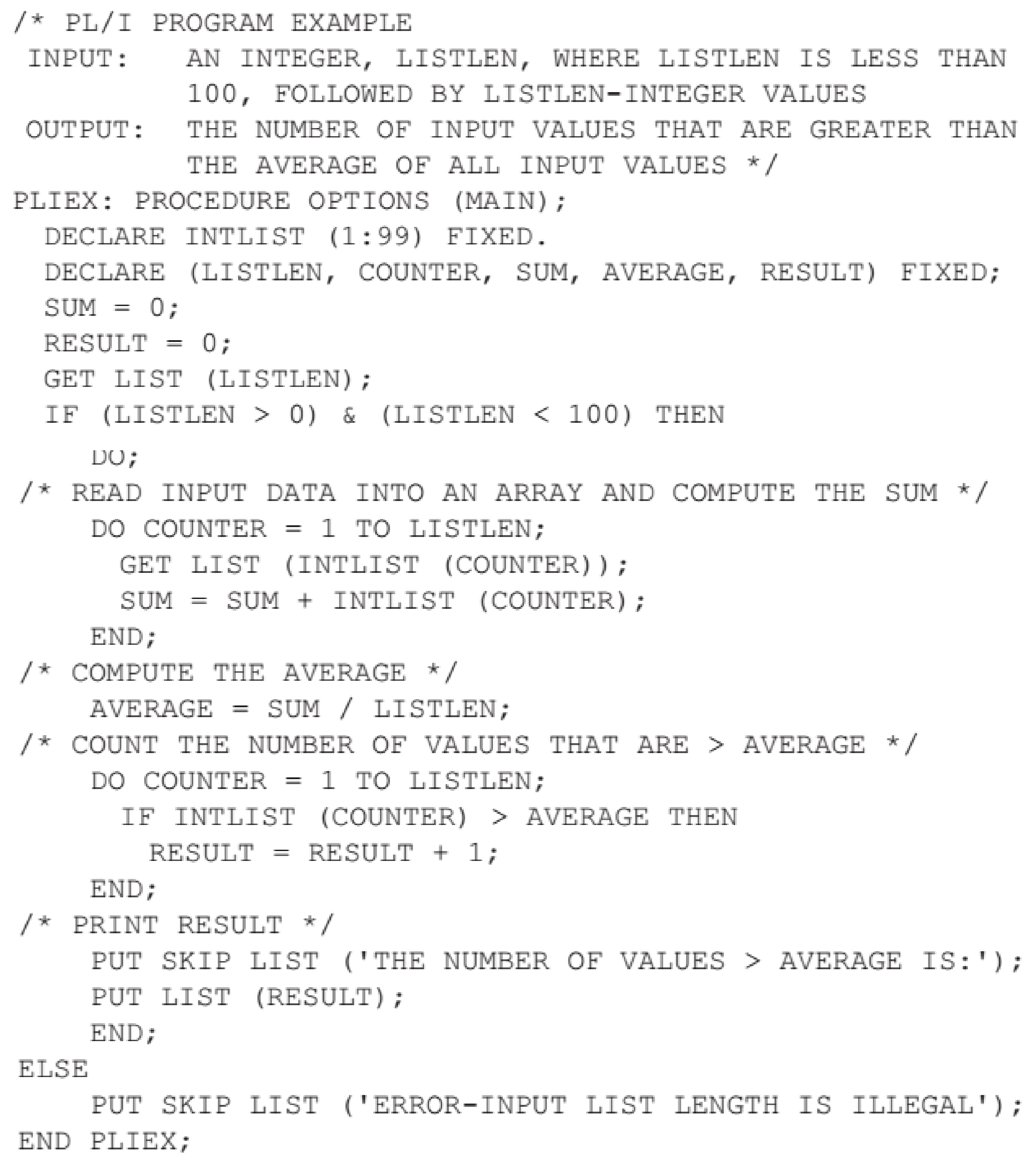
PL/I

- PL/I (Programming Language One) represents the first large-scale attempt to design a language that could be used for a broad spectrum of application areas.
- All previous and most subsequent languages have focused on one particular application area, such as science, artificial intelligence, or business.
- But, it is designed for use in both scientific and commercial applications.
- Initially called NPL (New Programming Language)
- Designed by IBM and SHARE in 1965.

PL/I

- Predecessors: Fortran IV, ALGOL 60 and COBOL.
- PL/I was the first programming language to have the following facilities:
  - Programs were allowed to create concurrently executing subprograms. Although this was a good idea, it was poorly developed in PL/I.
  - It was possible to detect and handle 23 different types of exceptions, or run-time errors.
  - Subprograms were allowed to be used recursively, but the capability could be disabled for efficiency.
  - Pointers were included as a data type.

An Example of a PL/I Program



```
/* PL/I PROGRAM EXAMPLE
 INPUT:    AN INTEGER, LISTLEN, WHERE LISTLEN IS LESS THAN
           100, FOLLOWED BY LISTLEN-INTEGER VALUES
 OUTPUT:   THE NUMBER OF INPUT VALUES THAT ARE GREATER THAN
           THE AVERAGE OF ALL INPUT VALUES */
PLIEX: PROCEDURE OPTIONS (MAIN);
  DECLARE INTLIST (1:99) FIXED.
  DECLARE (LISTLEN, COUNTER, SUM, AVERAGE, RESULT) FIXED;
  SUM = 0;
  RESULT = 0;
  GET LIST (LISTLEN);
  IF (LISTLEN > 0) & (LISTLEN < 100) THEN
     DO;
/* READ INPUT DATA INTO AN ARRAY AND COMPUTE THE SUM */
     DO COUNTER = 1 TO LISTLEN;
       GET LIST (INTLIST (COUNTER));
       SUM = SUM + INTLIST (COUNTER);
     END;
/* COMPUTE THE AVERAGE */
     AVERAGE = SUM / LISTLEN;
/* COUNT THE NUMBER OF VALUES THAT ARE > AVERAGE */
     DO COUNTER = 1 TO LISTLEN;
       IF INTLIST (COUNTER) > AVERAGE THEN
         RESULT = RESULT + 1;
     END;
/* PRINT RESULT */
     PUT SKIP LIST ('THE NUMBER OF VALUES > AVERAGE IS:');
     PUT LIST (RESULT);
     END;
ELSE
     PUT SKIP LIST ('ERROR-INPUT LIST LENGTH IS ILLEGAL');
END PLIEX;
```

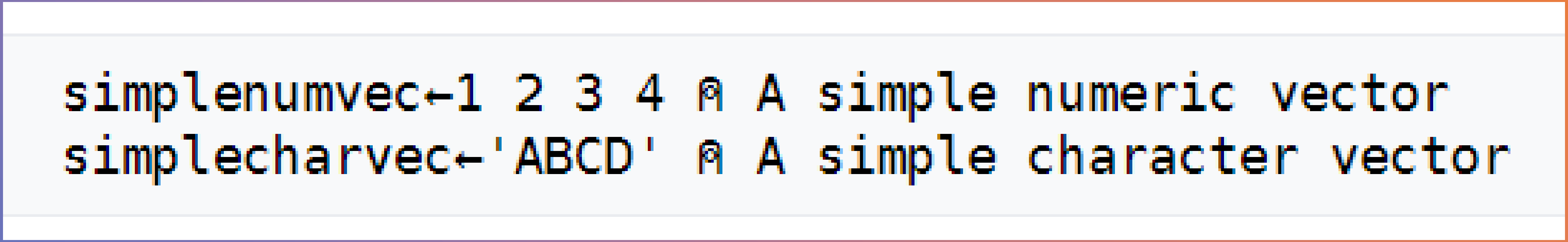
Two Early Dynamic Languages: APL and SNOBOL

Two Early Dynamic Languages: APL and SNOBOL

- The languages discussed here are <u>very different</u> from the others.
- Also, in appearance and in purpose, APL and SNOBOL are quite <u>different from each other</u>.
- They share two fundamental characteristics, however: <u>dynamic typing</u> and <u>dynamic storage allocation</u>.
- Variables in both languages are essentially <u>untyped</u>. A variable acquires a type <u>when it is assigned a value</u>, at which time it assumes the type of the value assigned.
- <u>Storage is allocated</u> to a variable only when it is assigned a value, because before that there is <u>no way to know</u> the amount of storage that will be needed.

APL

- APL was first described in the book from which it gets its name, **A Programming Language**. It was designed around 1960 by Kenneth E. Iverson at IBM.
- It was not originally designed to be an implemented programming language but rather was intended to be a vehicle for describing computer architecture (*a hardware description language*).
- Highly expressive (many operators, for both scalars and arrays of various dimensions)
- Programs are very difficult to read.
- Still in use; minimal changes.

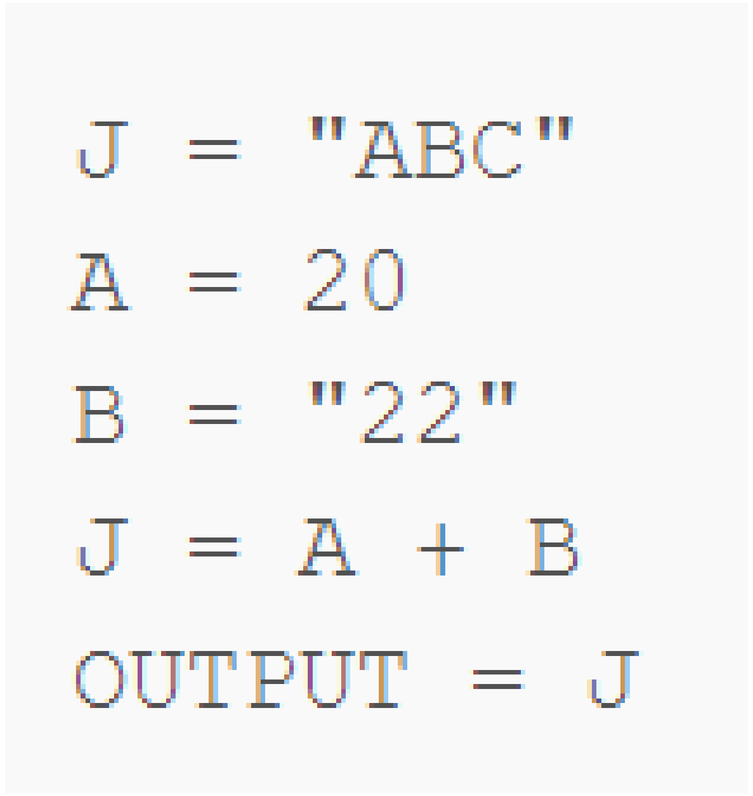


```
simplenumvec←1 2 3 4 ⍝ A simple numeric vector
simplecharvec←'ABCD' ⍝ A simple character vector
```

Example Array Definitions of APL

SNOBOL

- SNOBOL (**S**tri**N**g **O**riented and Sym**BO**lic **L**anguage) (pronounced “snowball”) was designed in the early 1960s by three people at Bell Laboratories: D. J. Farber, R. E. Griswold, and I. P. Polonsky.
- It was designed specifically for text processing (*a string manipulation language*).
- The heart of SNOBOL is a collection of powerful operations for string pattern matching.
- One of the early applications of SNOBOL was for writing text editors.
- Because the dynamic nature of SNOBOL makes it slower than alternative languages, it is no longer used for such programs.
- However, SNOBOL is still a live and supported language that is used for a variety of text-processing tasks in several different application areas.



```
J = "ABC"
A = 20
B = "22"
J = A + B
OUTPUT = J
```

A Sample Snobol Code

SNOBOL has dynamic typing. Variables do not need to be predeclared and their values can change types easily.

In the example, J starts off containing a string but then is changed so it contains an integer; 42 is printed.