Introduction

CS 315 – Programming Languages

Pinar Duygulu

Bilkent University

Course information

- Course Homepage

  www.cs.bilkent.edu.tr/~duygulu/Courses/CS315/

Reference Material

- Textbook:

  Robert W. Sebesta, *Concepts of Programming Languages*, Addison-Wesley, (Ninth Edition)
- Supplementary material:
  http://www.aw-bc.com/sebesta

Syllabus

- Preliminaries
- Syntax and Semantics
- Describing Syntax and Semantics
- Lexical and Syntax Analysis, Lex
- Lexical and Syntax Analysis, Yacc
- Names, Bindings, Type Checking, and Scopes
- Data Types
- Expressions and the Assignment Statement
- Statement-Level Control Structures
- Subprograms
- Implementing Subprograms
- Abstract Data Types
- Functional Programming Languages, Lisp
- Logic Programming Languages, Prolog

Grading

- Quizzes : 15%
- Homeworks : 15%
- Projects : 25%
- Midterm : 20%
- Final : 25%

This week's lecture : Preliminaries

- Why study concepts of programming languages?
- The major programming domains
- Criteria for language evaluation

Readings:

- Chapter 1 & 2
- C.A.R. Hoare: Hints on the Design of Programming Languages; Stanford Report

Why study programming languages?

- Working knowledge of one or two programming languages is not sufficient for a computer scientists

- You should study general concepts of language design and evaluation

Reasons for studying concepts of programming languages

- Increased capacity to express ideas
- Improved background for choosing appropriate languages
- Increased ability to learn new languages
- Better understanding of the significance of implementation
- Overall advancement of computing

Increased capacity to express ideas

It is difficult for people to conceptualize structures that they cannot describe verbally.

Consider natural languages. The depth at which you can think is influenced by the expressive power of the language that you are using to communicate.

It is similar for programming languages. The language in which you develop software puts limits on the kind of control structures, data structures and abstractions, and therefore the algorithms that you can develop.

Increased capacity to express ideas

Awareness of a wider variety of programming language features can reduce such limitations in software development.

If you have to use a specific language which does not have those capabilities still you can make use of those concepts by simulating them.

For example you can simulate associative arrays in Perl with the structures in C

Improved background for choosing appropriate languages

If all you have is a hammer, then everything looks like a nail

Same for PLs. If all you know is Java, then every solution to every problem will be a Java solution

Many programmer learn one or two languages specific to the projects.

Some of those languages may be no longer used

When they start a new project they continue to use those languages which are old and not suited to the current projects.

Improved background for choosing appropriate languages

However another language may be more appropriate

If they were familiar with the other languages, particularly the features in those languages they would choose better languages

Studying the principles of PLs provides a way to judge languages, and make informed statements: “The advantages of Perl for this problem are …..”, “The advantages of Java are ….”

Increased ability to learn new languages

- If you know the programming language concepts you will learn other languages much easier compared to programmers knowing only one or two languages.
- For example, if you know concept of object oriented programming, it is easier to learn C++ after learning Java
- Just like natural languages

  Learning new languages actually causes you to learn things about the languages you already know
- Example

  Languages may differ in the way in which arithmetic expressions are evaluated

  Doing something the old way in a new language and getting surprising results may cause you to double check your assumptions about the old language, which will hopefully help you to understand it better

Better understanding of the significance of implementation

- The best programmers are the ones having at least understanding of how things work under the hood
- You can simply write a code and let the compiler do everything, but knowing implementation details helps you to use a language more intelligently and write the code that is more efficient
- Also, it allows you to visualize how a computer executes language constructs (e.g. recursion is slow)

Increased ability to design new languages

- It is a very low possibility that you will design a general purpose programming language.
- However, you may end up designing a special purpose language to enter the commands for a software that you develop

Overall advancement of computing

New ways of thinking about computing, new technology, hence need for new appropriate language concepts

Not to repeat history

Although ALGOL 60 was a better language than FORTRAN, it did not become popular. It those who choose languages are better informed, better languages will be more popular.

Programming Domains

- Scientific Applications (first digital computers – 1940s)
  - Large floating-point arithmetic, execution efficiency, arrays and matrices, counting loops
  - **Examples:** FORTRAN, ALGOL 60, C
- Business Applications (1950s)
  - Producing elaborate reports, decimal numbers and character data
  - **Examples:** COBOL (1960s), Spread sheets, Word processors, Databases (SQL)
- Artificial Intelligence
  - Symbolic programming (names rather than numbers, linked lists rather than arrays)
  - **Examples**: LISP (1959), PROLOG (early 1970s)
- Systems Programming
  - System software : Operating system and all of the programming support tools of a computer system
  - Efficient and fast execution, low-level features for peripheral device drivers
  - **Examples**: PLS/2 (IBM Mainframe), BLISS (Digital), C (Unix)
- Scripting Languages
  - List of commands (Script) to be executed is put in a file.
  - **Examples:** sh, csh, tcsh, awk, gawk, tcl, perl, javascript
- Special-Purpose lnguages
  - **Examples:** RPG (Business Reports), SPICE (Simulation of Electronic Circuitry), SPSS (Statistics), Latex, nroff, troff (Document preparation packages). HTML, XML (internet programming)

Language Evaluation Criteria

To examine the underlying concepts of the various constructs and capabilities of programming languages

- Readability
- Writability
- Reliability
- Cost

Readability

Ease with which programs can be read and understood

in the early times, efficiency and machine readability was important
1970s-Software life cycle: coding (*small*) + maintenance (*large*)
Readability is important for maintenance

Characteristics that contribute to readability:

- Overall simplicity
- Orthogonality
- Control statements
- Data types and structures
- Syntax Considerations

Overall simplicity

- Large number of basic components - difficult to learn
- User learns only a subset
  - but this subset may differ from one user to another
- feature multiplicity:having more than one way to accomplish an operation
  - e.g. In Java

```
count = count + 1
count += 1
count ++
++count
```

- operator overloading
  - if users are allowed to create their own and not use this sensibly it is a problem, e.g. to use + for integer and floating point addition is acceptable, but to sum up all the elements of two singe dimensional arrays is not – different from vector addition

- On the other hand, the simplest does not mean the best. e.g.Assembly languages

Orthogonality

- It means that a relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the language
- Furthermore, every possible combination of primitives is legal and meaningful

Example:

- Four primitive data types : integer, float, double and character
- Two type operators : array and pointer
- If the two type operators can be applied to themselves and the four primitive data types, a large number of data structures can be defined
- However, if pointers were not allowed to point to arrays, many of those possibilities would be eliminated

Orthogonality

- Example : Adding two 32-bit integers residing in memory or registers, and replacing one of them with the sum

- IBM (Mainframe) Assembly language has two instructions:

  A Register1, MemoryCell1

  AR Register1, Register2

  They mean

  Register1 ← contents(Register1) + contents(MemoryCell1)

  Register1 ← contents(Register1) + contents(Register2)

- WAX Assembly language has one instruction:

  ADDL operand1, operand2

  It means

  operand2 ← contents(operand1) + contents(operand2)

  Here, either operand can be a register or a memory cell.

Orhthogonality

- VAX instruction design is orthogonal
  - Because a single instruction can use either registers or memory cells as the operands
- IBM design is not orthogonal
  - Only two operand combinations are legal out of for possibilities and two require different instructions, A and AR
  - More restricted and less writable (you cannot add two values and store the sum in memory location)
  - More difficult to learn because of the restrictions and the additional instructions

Orthogonality

- Orthogonality is closely related to simplicity
- The more orthogonal the design of a language, the fewer exceptions the language rules require

- **Pascal is not an orthogonal** language, because

  A function cannot return a record (only unstructured types allowed),

  A file must be passed as a **`var`** parameter,

  Formal parameter types must be named (cannot be type descriptions)

  Compound statements are formed by **`begin-end`** pair, except repeat-until

- **C is not an orthogonal language, because**

  -records(structs) can be returned from functions but arrays cannot

  -a member of a structure can be any type but not void or structure of the same type

  -a member of an array can be any type but not void or function

Orthogonality

Too much orthogonality can cause problems as well:

ALGOL68 is the most orthogonal language.

- Every construct has a type
- Most constructs produce values
- This may result in extremely complex constructs,
- e.g., A conditional can appear as the left side of an assignment statement, as long as it produces a location:

```
(if (A<B) then C else D) := 3
```

- This extreme form of orthogonality leads to unnecessary complexity

Orthogonality

Functional languages offer a good combination of simplicity and orthogonality.

Control Statements

- Control statement design of a language is an important factor in readability.

- For example, languages such as BASIC and FORTRAN, that allow the use of `goto` statement, are **not readable**.

Control Statements

```
while (incr < 20) {
  while (sum < =100) {
   sum += incr;
  }
  incr++;
}
```

Control Statements

If C did not have a loop construct:

```
loop1:
  if (incr >=20) go to out;
loop2:
  if (sum > 100) go to next;
   sum += incr;
   go to loop2;
next:
  incr++;
  go to loop1:
out:
```

Data types and structures

Facilities for defining data types and data structures are helpful for readability

- If there is no boolean type available then a flag may be defined as integer:

  `found = 1` (instead of `found = true`)

  May mean something is found as boolean or what is found is 1

An array of record type is more readable than a set of independent arrays

- In Fortran

```
Character(Len=30) Name(100)
IntegerEmployeeNumber(100)
Real Salary(100)
```

Syntax considerations

- **Identifier Forms:** restricting identifier length is bad for readability.
- Example:
  - FORTRAN77 identifiers can have at most **6** characters.
  - The extreme case is the ANSI BASIC, where an identifier is either a single character or a single character followed by a single digit.

- Availability of word concatenating characters (e.g., _) is good for readability.

Syntax considerations

Special Words:

Readability is increased by special words (e.g., **`begin`**, **`end`**, **`for`**).

In PASCAL and C, **`end`** or **`}`** is used to end a compound statement. It is difficult tell what an **`end`** or **`}`** terminates.

However, ADA uses **`end if`** and **`end loop`** to terminate a selection and a loop, respectively.

Another issue is the use of special words as names of variables. For example, in FORTRAN77, special words, e.g., **`DO`** and **`END`** can be used as variable names.

Syntax considerations

Forms and Meaning:

Forms should relate to their meanings. Semantics should directly follow from syntax.

For example,

sin(x)

should be the sine of **x**,

not the sign of **x** or cosign of **x**.

- Grep is hard to understand for the people not familiar with using regular expressions
- grep : g/regular_expression/p /reg_exp/ : search for that reg_exp g: scope is whole file p:print

Writability

- Ease of creating programs
- Characteristics that contribute to readability
  - Simplicity and Orthogonality
  - Support for abstraction
  - Expressivity

Simplicity and orthogonality

Simplicity and orthogonality are good for writability also.

- When there are large number of construct programmers may not be familiar with all of them, and lead to either misuse or disuse of those items.
- A smaller number of primitive constructs (simplicity) and consistent set of rules for combining them (orthogonality) is good for writability
- However, too much orthogonality may lead to undetected errors, since almost all combinations are legal.

Support for abstraction

Abstraction: ability to define and use complicated structures and operations in ways that allows ignoring the details.

Abstraction is the key concept in contemporary programming languages

The degree of abstraction allowed by a programming language and the naturalness of its expressions are very important to its writability.

PLs can support two types of abstraction:

process

data

Process abstraction

The simplest example of abstraction is **subprograms** (e.g., methods).

You define a subprogram, then use it by ignoring how it actually works.

Eliminates replication of the code

Ignores the implementation details

e.g. sort algorithm

Data abstraction

As an example of **data abstraction**, a tree can be represented more naturally using pointers in nodes.

In FORTRAN77, where pointer types are not available, a tree can be represented using 3 parallel arrays, two of which contain the indexes of the offspring, and the last one containing the data.

Expressivity

Having more convenient and shorter ways of specifying computations.

For example, in C,

```
count++;
```

is more convenient and expressive than

```
count = count + 1;
```

`for` is more easier to write loops than `while`

Reliability

- Reliable: it performs to its specifications under all conditions
  - Type Checking
  - Exception Handling
  - Aliasing
  - Readability and writability

Type Checking

- Testing for type errors in a given program either by the compiler or during program execution

- The compatibility between two variables or a variable and a constant that are somehow related (e.g., assignment, argument of an operation, formal and actual parameters of a method).

- **Run-time** (Execution-time) checking is expensive.
- **Compile-time** checking is more desirable.

- The earlier errors in programs are detected, the less expensive it is to make the required repairs

Type Checking

Ada requires type checking on all variables in compile time.

Original C language requires no type checking neither in compilation nor execution time. That can cause many problems.

Java requires checks of the types of nearly all variables and expressions at compile time

Type Checking

For example, the following program compiles an runs!

```
foo (float a) {
  printf ("a: %g and square(a): %g\n", a,
  a*a);
}
main () {
  char z = 'b';
  foo(z);
}
```

Output is : `a: 98 and square(a): 9604`

Type Checking

- **Subscript range** checking for arrays is a part of type checking, but it must be done in the **run-time**. Out-of-range subscript often cause errors that do not appear until long after actual violation.

Exception Handling

- The ability of a program
- to intercept run-time errors, as well as other unusual conditions
- to take corrective measures and continue

- Ada, C++, and Java include extensive capabilities for exception handling, but in C and Fortran it is practically non-exsistent

Aliasing

- Having two distinct referencing methods (or names) for the same memory cell.
- It is a dangerous feature in a programming language.
- E.g., pointers in PASCAL and C
- two different variables can refer to the same memory cell

Readability and Writability

- The easier a program **to write**, the more likely it is **correct**.
- Programs that are difficult to read are difficult both to write and modify.

Cost

1)Cost of **training** the programmers. Function of simplicity and orthogonality, experience of the programmers.

2)Cost of **writing** programs. Function of the writability

-These two costs can be reduced in a good programming environment

3)Cost of **compiling** programs (cost of compiler, and time to compile)

Cost

4)Cost of **executing** programs. If a language requires many run-time type checking, the programs written in that language will execute slowly.

Trade-off between compilation cost and execution cost.

Optimization: decreases the size or increases the execution speed.

Without optimization, compilation cost can be reduced.

Extra compilation effort can result in faster execution.

Cost

5) Cost of the implementation system. If expensive or runs only on expensive hardware it will not be widely used

6) Cost of reliability – important for critical systems such as a power plant or X-ray machine

7) Cost of **maintaining** programs. For corrections, modifications and additions. Function of readability.

   Usually, and unfortunately, maintenance is done by people other that the original authors of the program

   For large programs, the maintenance costs is about **2** to **4** times the development costs.

Other criteria for evaluation

Portability: program can be moved from one environment to another

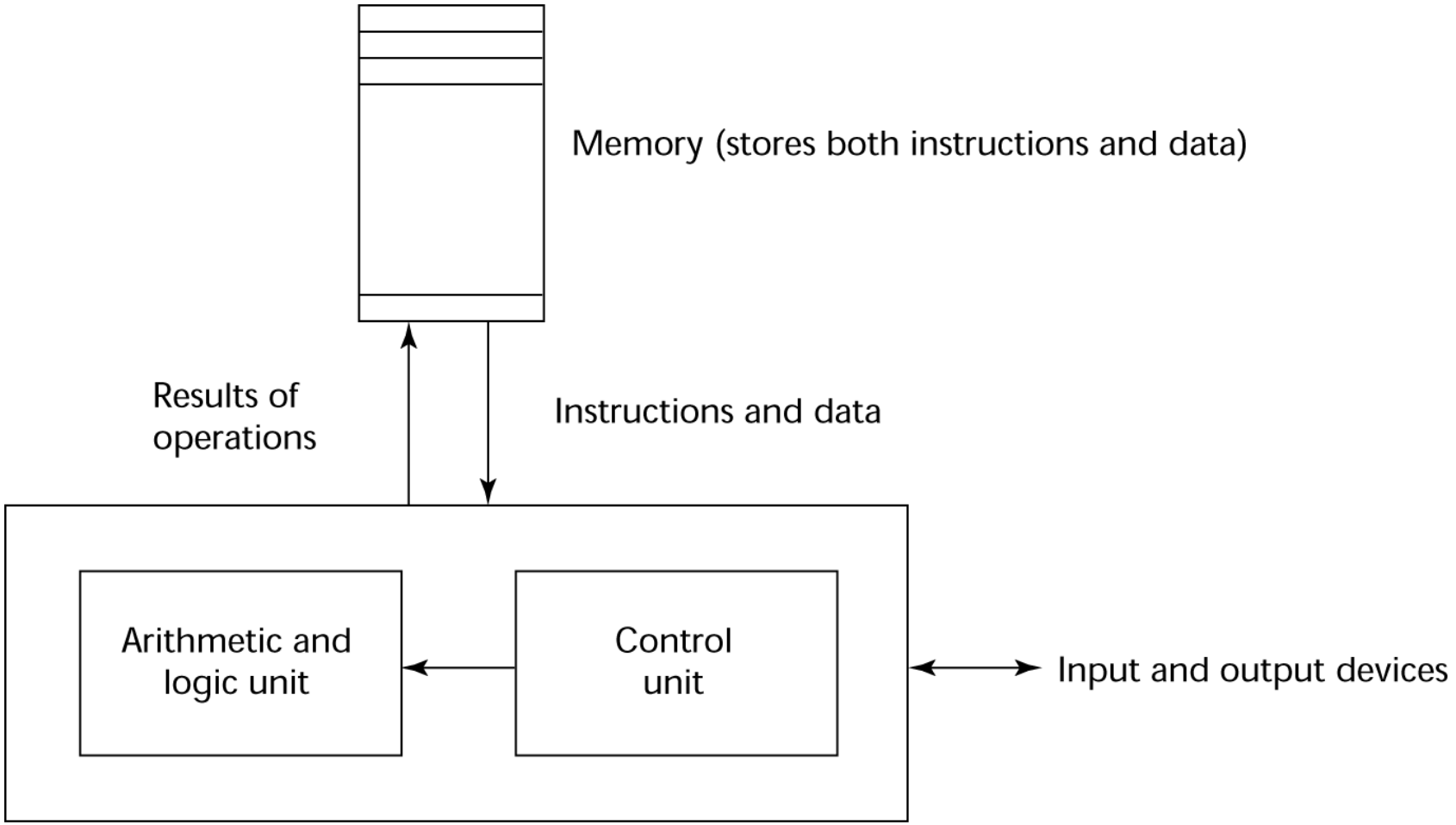
Generality: applicable to wide range of applications

Well-definedness: complete and precise definition of the language

Influence on Language Design

- Several other factors influence the design of PLs
  - Computer architecture
  - Programming Methodologies

The von Neumann computer architecture



Computer Architecture

- **Von Neumann architecture ⇒ Imperative languages**
- CPU executes instructions ⇒ operations, statements
- Operands reside in the memory cells ⇒ variables
- Values are stored back to memory ⇒ assignment
- Branching, jumping ⇒ **`goto`**, **`for`**, **`repeat`**, **`while`** loops (iteration)

- Discourages **recursion**, although it is natural.
- Functional languages are not efficient on Von Neumann architectures.
- Several attempts were made on new architectures for LISP (LISP machine, and Symbolics). They did not survive.

Program Design Methodologies

- There has been a major shift from **process-oriented** programming to **data-oriented** programming.
- More emphasis on **Abstract Data Types**.
- The latest trend in the evaluation of data-oriented software development is the **object-oriented** design. It is based on Data Abstraction, Encapsulation, Information hiding, Dynamic type binding.
- Smalltalk, Objective C, C++, Java are O-O programming languages.

Language Categories

- PL's are often categorized into four categories:
- Imperative, Functional, Logic, and Object-Oriented.

- Visual languages

Language Design tradeoffs

- The programming language **evaluation criteria are conflicting**.
- **reliability vs cost of execution :** in Java all references to array elements are checked to ensure that the indices are in their legal ranges. C does not require such a checking
- **writability vs readability:** in APL there are many powerful sets of operators for array operands, very short programs can be written but it is hard to read those programs
- **flexibility vs safety:** variant records in PASCAL that allow a memory cell to contain different values at different times

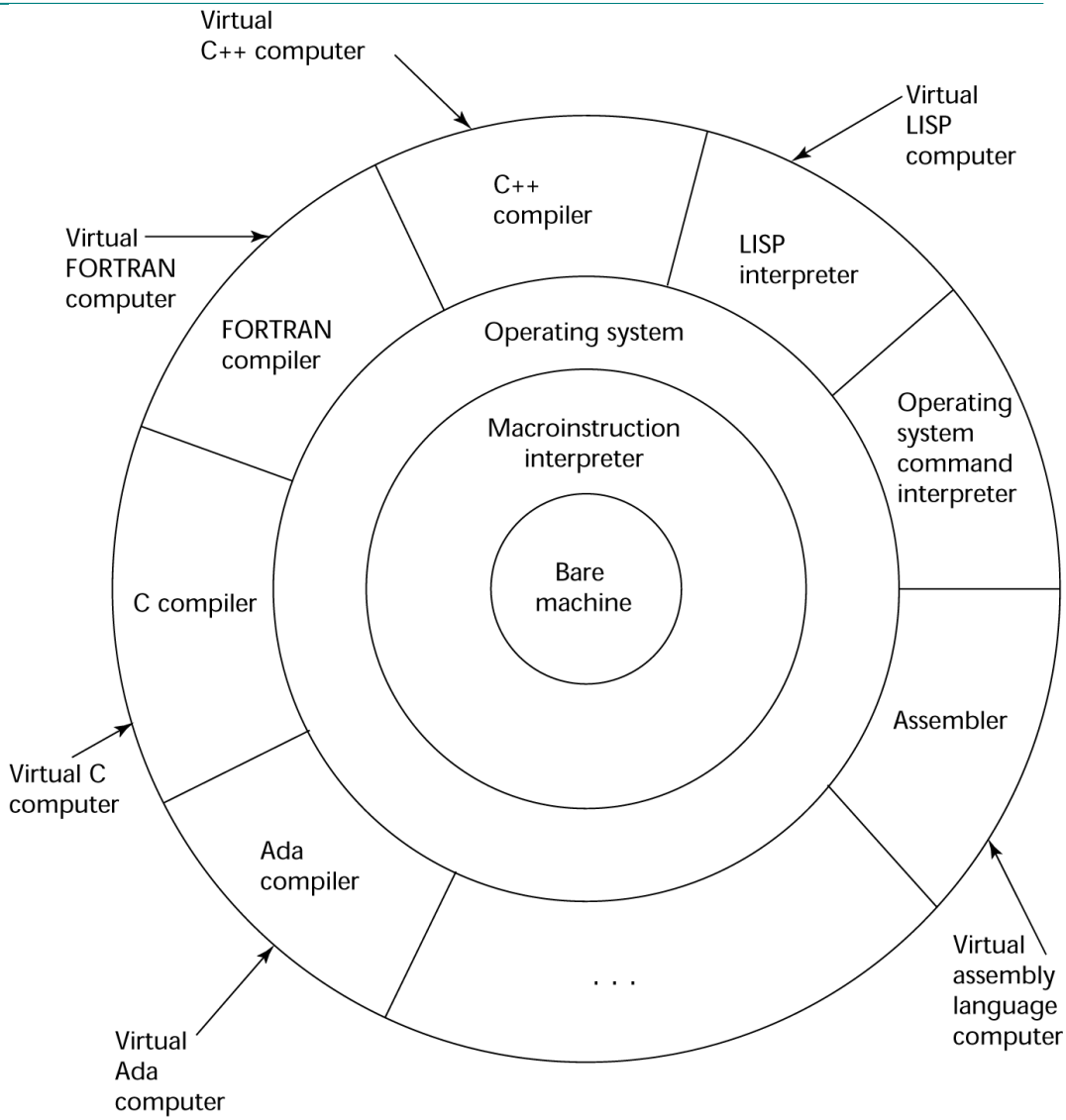
Layered interface of virtual computers, provided by a typical computer system

Bare Machine: Internal memory and processor

Macroinstructions: Primitive operations, or machine instructions such as those for arithmetic and logic operations

Operating system: Higher level primitives than machine code, which provides system resource management, input and output operations, file management system, text and/or program editors…

Virtual computers: provides interfaces to user at a higher level



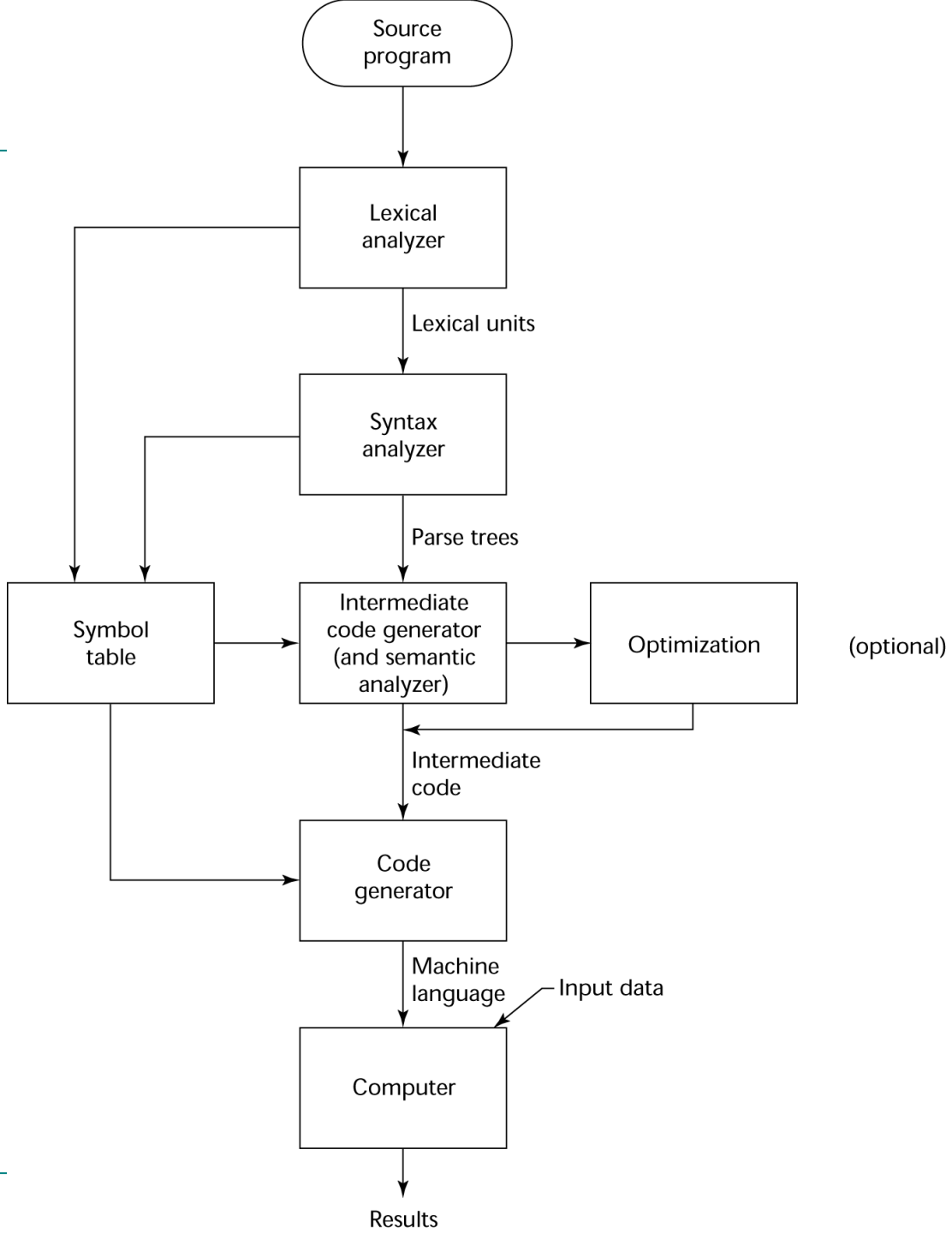
Implementation

- Compilation
- Pure Interpretation
- Hybrid implementation

Compilation

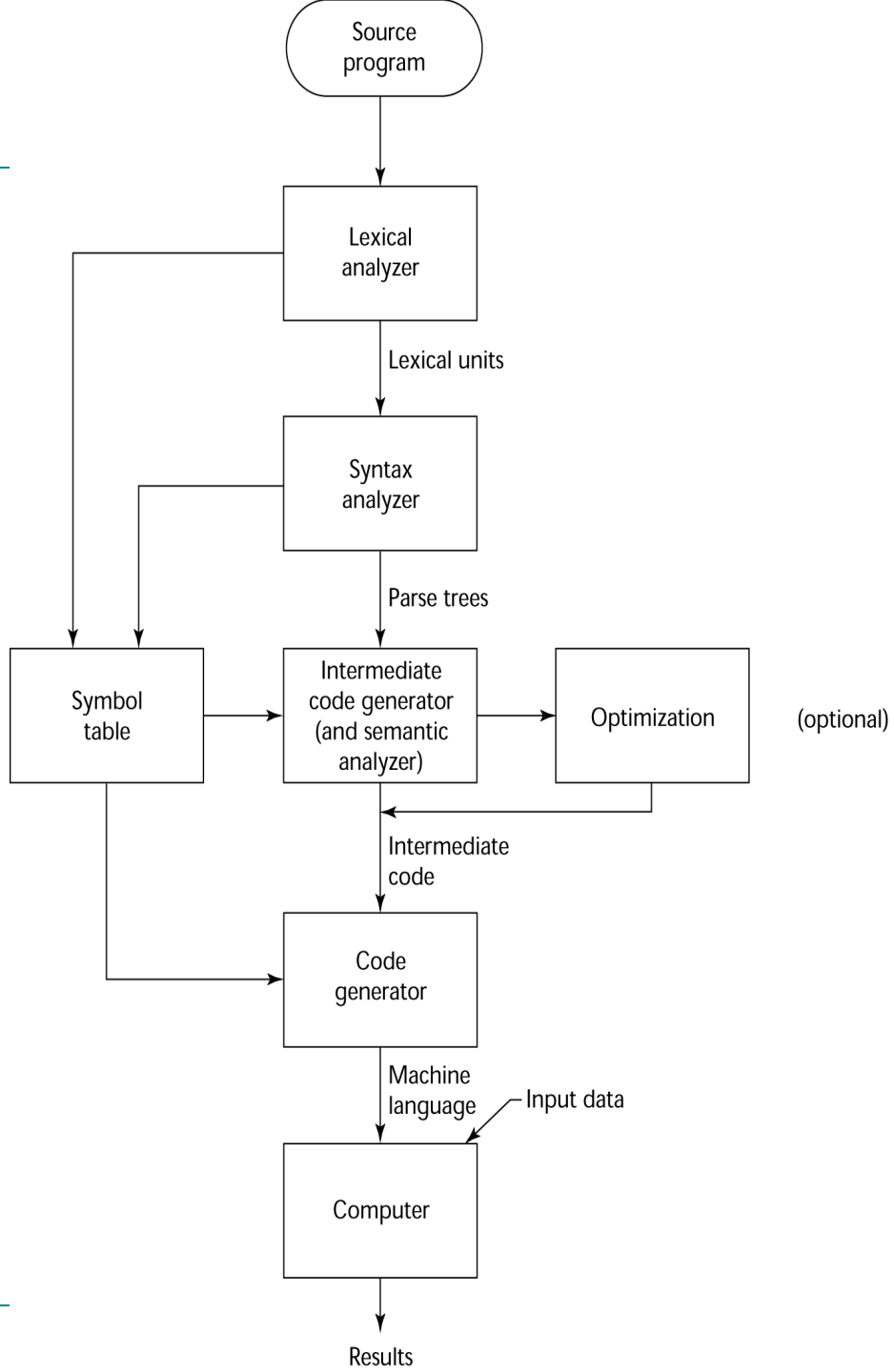
- A program is translated to machine code, which is directly executed on the computer.
- **Advantage**: very fast program execution, once the translation process is complete

- E.g. C, COBOL, Ada



Compilation

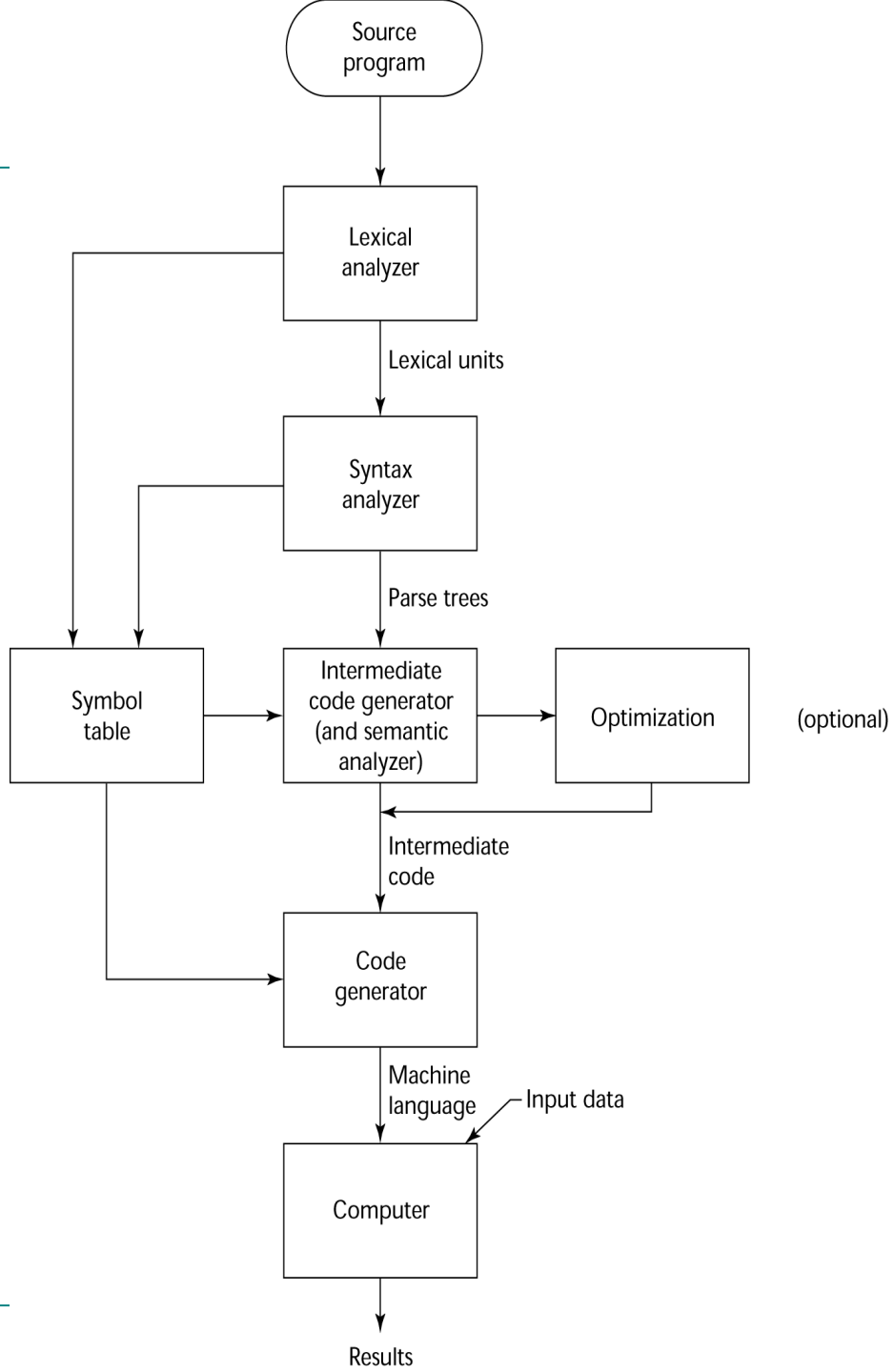
- **Source language:** The language that the compiler translates
- **Lexical analyzer:** gathers the characters of the source program into lexical units (e.g. identifiers, special words, operators, punctuation symbols) Ignores the comments
- **Syntax analyzer:** takes the lexical units, and use them to construct parse trees, which represent the the syntactic structure of the program
- **Intermediate code generator:** Produces a program at an intermediate level. Similar to assembly languages



Compilation

- **Semantic analyzer:** checks for errors that are difficult to check during syntax analysis, such as type errors
- **Optimization :** optional

  Used if execution speed is more important than compilation speed
- **Code generator:** Translates the intermediate code to machine language program
- **Symbol table:** serves as a database of type and attribute information of each user defined name in a program. Placed by lexical and syntax analyzers, and used by semantic analyzer and code generator



Linker

- Most user programs require programs from the operating system, such as input/output
- Before the machine language programs produced by a compiler can be executed, the required programs from the operating system must be found and linked to user programs
- The linking operation connects the user program to the system programs by placing the addresses of the entry points of the system programs in the calls to them in the user programs
- User programs must often be linked to previously compiled user programs that reside in libraries

Fetch-Execute cycle

- Execution of machine code program on a von Neumann architecture computer occurs in a process called the **fetch-execute cycle**
- Programs reside in memory but are executed in the CPU
- Each instruction to be executed must be moved from memory to CPU
- The address of the next instruction to be executed is maintained in a register called the **program counter**

Fetch-Execute cycle

```
Initialize the program counter
repeat forever
   fetch the instruction pointed to by the program counter
   increment the program counter to point at the next instruction
   decode the instruction
   execute the instruction
End repeat
```

Program terminates when stop instruction is encountered

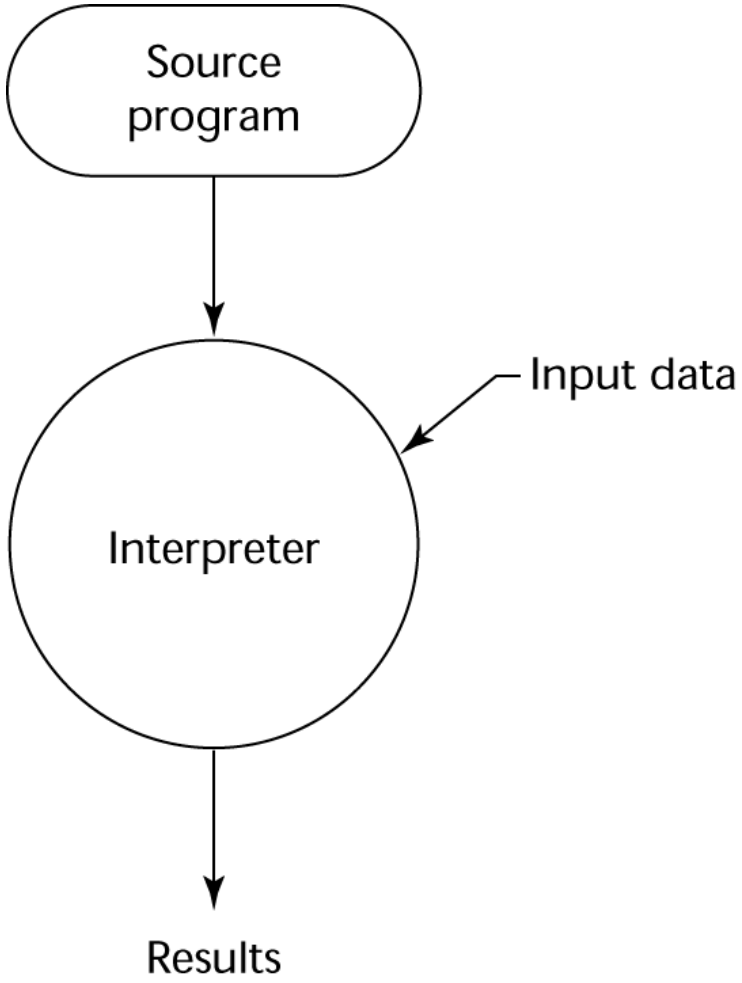
Or, control transfers from user program to OS when the user program execution is complete

Von Neumann Bottleneck

- The speed of the connection between a computer's memory and its processors determines the speed of the computer

Pure interpretation

- Each statement is decoded and executed individually.
- No translation
- The interpreter program acts as a software simulation of a machine whose fetch-execute cycle deals with high level language program statements rather than machine instructions
- **Advantage**: debugging is easy. Good for development
- **Disadvantage**: slow execution.
- It must be decoded everytime



Compiler vs Interpreter

How an English speaker can communicate to a French speaker

1. The English speaker employs an *interpreter* who translates the English sentences into French as they are spoken. The English speaker says a sentence in English, the interpreter hears it, translates it in his/her brain into French, and the speaks the sentence in French. This is repeated for each sentence. Progress is slow: there are pauses between sentences as the translation process takes place.

1. The English speaker writes down (in English) what needs to be said. The whole document is then translated into French, producing another piece of paper with the French version written on it. There are two factors to consider in this scenario:
   - There is a slight delay at the start as the English document needs to be translated in its entity before it can be read.
   - The translated document can be read at any time after that, and at the normal speed at which the French-speaker reads.

Compiler vs Interpreter

A **compiler** translates a complete source program into machine code. The whole source code file is compiled in one go, and a complete, compiled version of the file is produced. This can be saved on some secondary storage medium (e.g. floppy disk, hard disk...). This means that:

The program can only be executed once translation is complete

ANY changes to the source code require a complete recompilation.

An **interpreter**, on the other hand, provides a means by which a program written in source language can be understood and executed by the CPU line by line. As the first line is encountered by the interpreter, it is translated and executed. Then it moves to the next line of source code and repeats the process. This means that:

The interpreter is a program which is loaded into memory alongside the source program

Statements from the source program are fetched and executed one by one

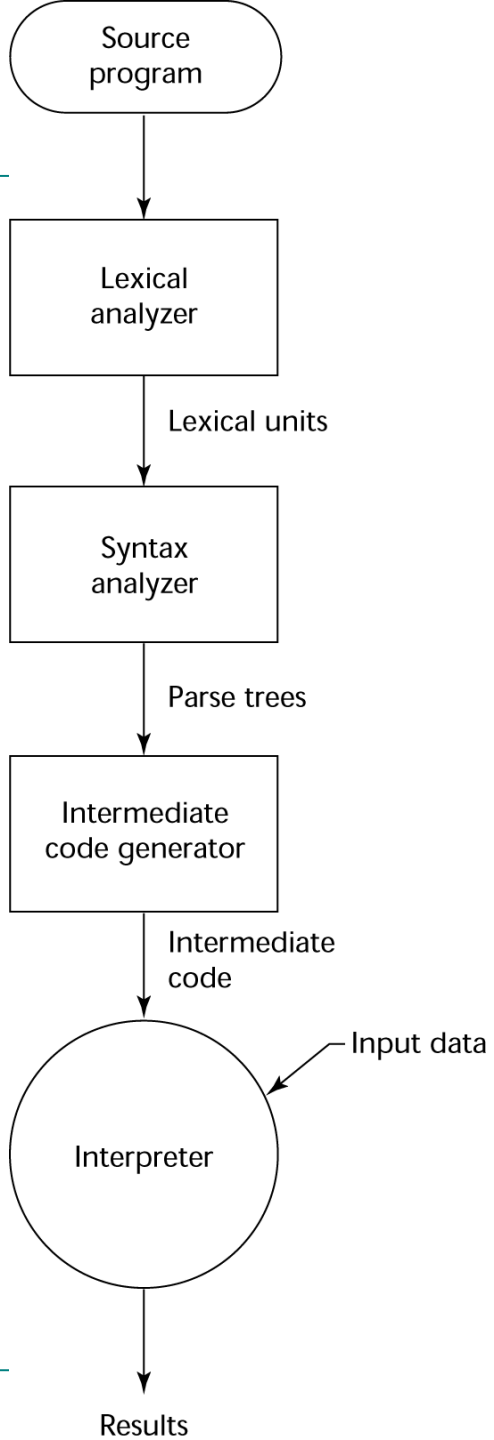
No copy of the translation exists, and if the program is to be re-run, it has to be interpreted all over again.

Compiler vs Interpreter

- Because the interpreter must be loaded into memory, there is less space available during execution; a compiler is only loaded into memory during compilation stage, and so only the machine code is resident in memory during run-time;
- Once we have a compiled file, we can re-run it any time we want to without having to use the compiler each time; With any interpreted language, however, we would have to re-interpret the program each time we wanted to run it;
- Machine code programs run more quickly than interpreted programs;
- However, it is often quicker and easier to make changes with an interpreted program than a compiled one, and as such development time may be reduced.

Hybrid implementation system

- Translate high-level language programs to and intermediate language designed to allow easy interpretation.



Compiler vs Interpreter in Java

•In Java intermediate form is called byte code, which provides portability to any machine that has a byte code interpreter and an associated run-time system. Together these are called Java Virtual Machine

Typically, when used in that generic manner, the term *Java **compiler*** refers to a program which translates Java language source code into the Java Virtual Machine (JVM) *bytecodes*. The term *Java **interpreter*** refers to a program which implements the JVM specification and actually executes the bytecodes (and thereby running your program).

Interpreter and compiler

•If both interpreter and compiler exists for a language (e.g., LISP), programs are first developed using the interpreter, then they are compiled to obtain fast executing programs.

Programming environments

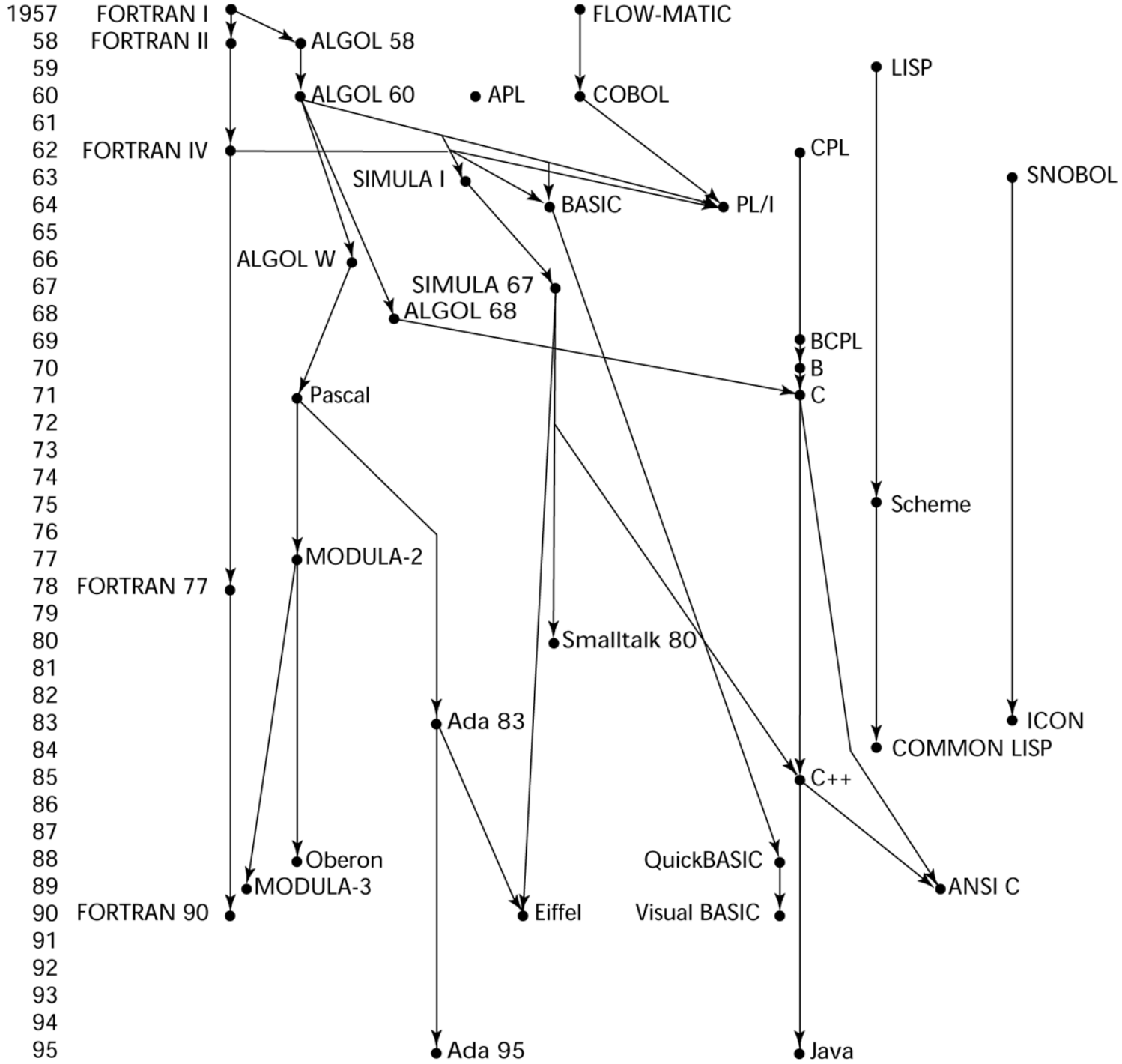
- Text editor, pretty-printer, linker, compiler, debugger, graphical interface, interaction between these tools.
- JCreator is an example of a programming environment.

History

Most of the ideas of Modern PLs appear in four or five classic languages

- Fortran (1956/8): Jump-based control structures (looping, conditionals) subroutines, arrays, formatted I/O
- Cobol: Task-specific types for business applications, e.g. decimal arithmetic and strings
- Algol (1960+): Lexical scoping, composite types(records), automated type-checking, high-level control structures, recursion
- Lisp(1959): Functions yielding functions as return values, same notation for code and data, dynamic typing, recursion in lieu of iteration
- Simula(1967): Information hiding, object oriented programming

History


1957
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
FORTRAN I
FORTRAN II
ALGOL 58
FLOW-MATIC
LISP
ALGOL 60
APL
COBOL
FORTRAN IV
CPL
SNOBOL
SIMULA I
BASIC
PL/I
ALGOL W
SIMULA 67
ALGOL 68
BCPL
B
C
Pascal
Scheme
MODULA-2
FORTRAN 77
Smalltalk 80
Ada 83
ICON
COMMON LISP
C++
Oberon
QuickBASIC
MODULA-3
ANSI C
FORTRAN 90
Eiffel
Visual BASIC
Ada 95
Java

‘Hello World’ in different languages

http://www.all-science-fair-projects.com/science_fair_projects_encyclopedia/Hello_world_program

Java

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

C++

```
#include <iostream>
int main()
{
    std::cout << "Hello, world!\n";
}
```

C

```
#include <stdio.h>
int main()
{
   printf("Hello, world!\n");
   return 0;
}
```

Fortran

```
      PROGRAM
      HELLO
      WRITE(*,10)
10    FORMAT('Hello, world!')
      STOP
      END
```

LISP

```
(format t "Hello world!~%")
```

COBOL

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO-WORLD.

ENVIRONMENT DIVISION.

DATA DIVISION.

PROCEDURE DIVISION.
DISPLAY "Hello, world!".
STOP RUN.
```

PERL

```
print "Hello, world!\n";
```

PROLOG

```
write('Hello world'),nl.
```

ADA

```
with Ada.Text_Io;
procedure Hello is
begin
      Ada.Text_Io.Put_Line ("Hello, world!");
end Hello;
```

Assembly Language

```
bdos    equ 0005H   ; BDOS entry point
start:  mvi c,9     ; BDOS function: output string
lxi     d,msg$      ; address of msg
call    bdos ret    ; return to CCP

msg$:   db 'Hello, world!$'
end     start
```

HTML

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<title>Hello, world!</title>
<meta http-equiv="Content-Type“ content="text/html;
charset=UTF-8">
</head>
<body>
 <p>Hello, world!</p>
 </body>
</html>
```