Principles of Programming Language

Introduction

Policy

- **Plagiarism in assignments/quiz/exams will not be tolerated and will be strictly dealt with**
- Absence in quiz/exams for genuine reasons will have to be informed in advance
- Attendance will be recorded for every lecture. Minimum 75% attendance is mandatory.

Abstraction

"The process of removing physical, spatial, or temporal details or attributes in the study of objects or systems to focus attention on details of greater importance"

[Wikipedia]

This Course is not about…

- teaching you a programming language
- teaching you how to program

This Course is to…

- Introduce fundamental concepts of programming languages
- Discuss design issues of various language constructs
- Examine design/implementation choices for these constructs
- Compare design alternatives

Why Study PPL?

- To improve your ability to develop effective algorithms and to use your language
- Increased ability to learn new languages
- To allow a better choice of PL

Tiobe Statistics

Jan 2023	Jan 2022	Change	Programming Language	Ratings	Change
1	1		Python	16.36%	+2.78%
2	2		C	16.26%	+3.82%
3	4	↑	C++	12.91%	+4.62%
4	3	↓	Java	12.21%	+1.55%
5	5		C#	5.73%	+0.05%
6	6		Visual Basic	4.64%	-0.10%
7	7		JavaScript	2.87%	+0.78%
8	9	↑	SQL	2.50%	+0.70%
9	8	↓	Assembly language	1.60%	-0.25%
10	11	↑	PHP	1.39%	-0.00%
11	10	↓	Swift	1.20%	-0.21%
12	13	↑	Go	1.14%	+0.10%
13	12	↓	R	1.04%	-0.21%
14	15	↑	Classic Visual Basic	0.98%	+0.01%
15	16	↑	MATLAB	0.91%	-0.05%
16	18	↑	Ruby	0.80%	-0.08%
17	14	↓	Delphi/Object Pascal	0.73%	-0.27%
18	26	⇈	Rust	0.61%	+0.11%
19	20	↑	Perl	0.59%	-0.12%
20	23	↑	Scratch	0.58%	-0.01%

Why Study PPL?

- To improve your ability to develop effective algorithms and to use your language
- Increased ability to learn new languages
- To allow a better choice of PL
- To understand significance of implementation
- To make it easier to design a new language

Models of Programming Languages

1) Imperative

2) Functional

3) Logic

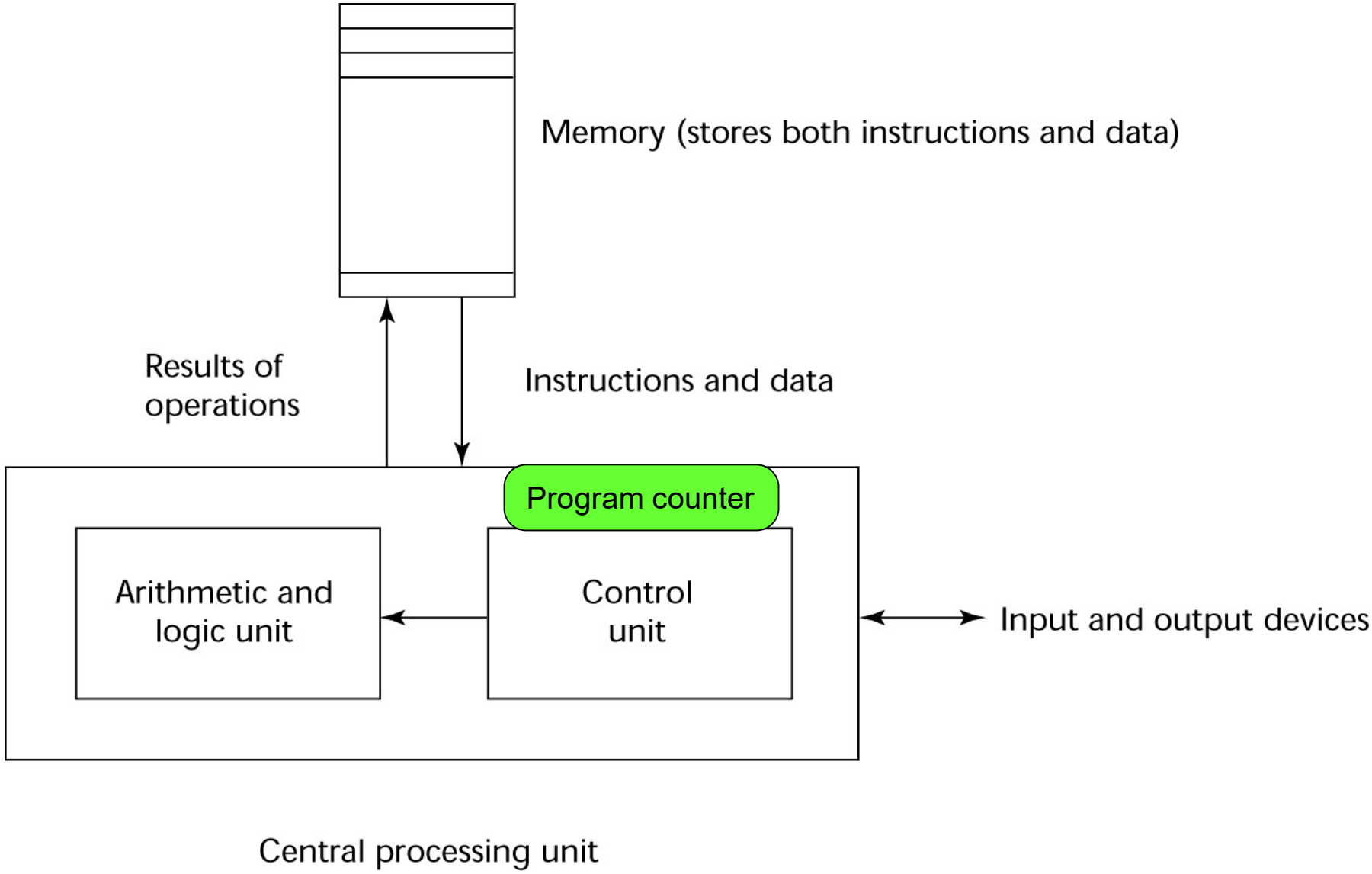
1) Imperative

- Computers take commands and do operations
- Thus, programming is like issuing procedural commands to the computer

Example: a factorial function in C

```
int fact(int n) {
  int factorial = 1;
  while (n>0) factorial *= n--;
  return factorial;
}
```

The von Neumann Architecture



1) Imperative

Imperative languages mimic von Neumann architecture

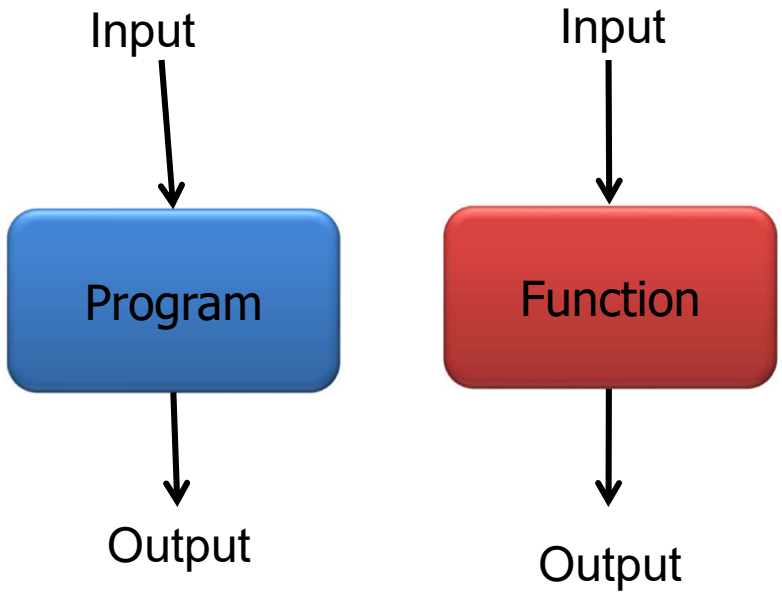
Variables ←→ memory cells
Assignment statements ←→ data piping between memory and CPU
Operations and expressions ←→ CPU executions
Explicit control of execution flows ←→ program counter

2) Functional

Programming is like solving mathematical functions, e.g.,
z = f(y, g(h(x)))
A program, and its subprograms, are just implementations of mathematical functions

Example: a factorial function in ML

```
fun fact x =
  if x <= 0
  then 1
  else x * fact(x-1);
```



3) Logic

- Program expressed as rules in formal logic
- Execution by rule resolution

Example: relationship among people

fact:
```
mother(joanne,jake).
father(vern,joanne).
```

rule:
```
grandparent(X,Z) :- parent(X,Y),
                    parent(Y,Z).
```

goal:
```
grandparent(vern,jake).
```

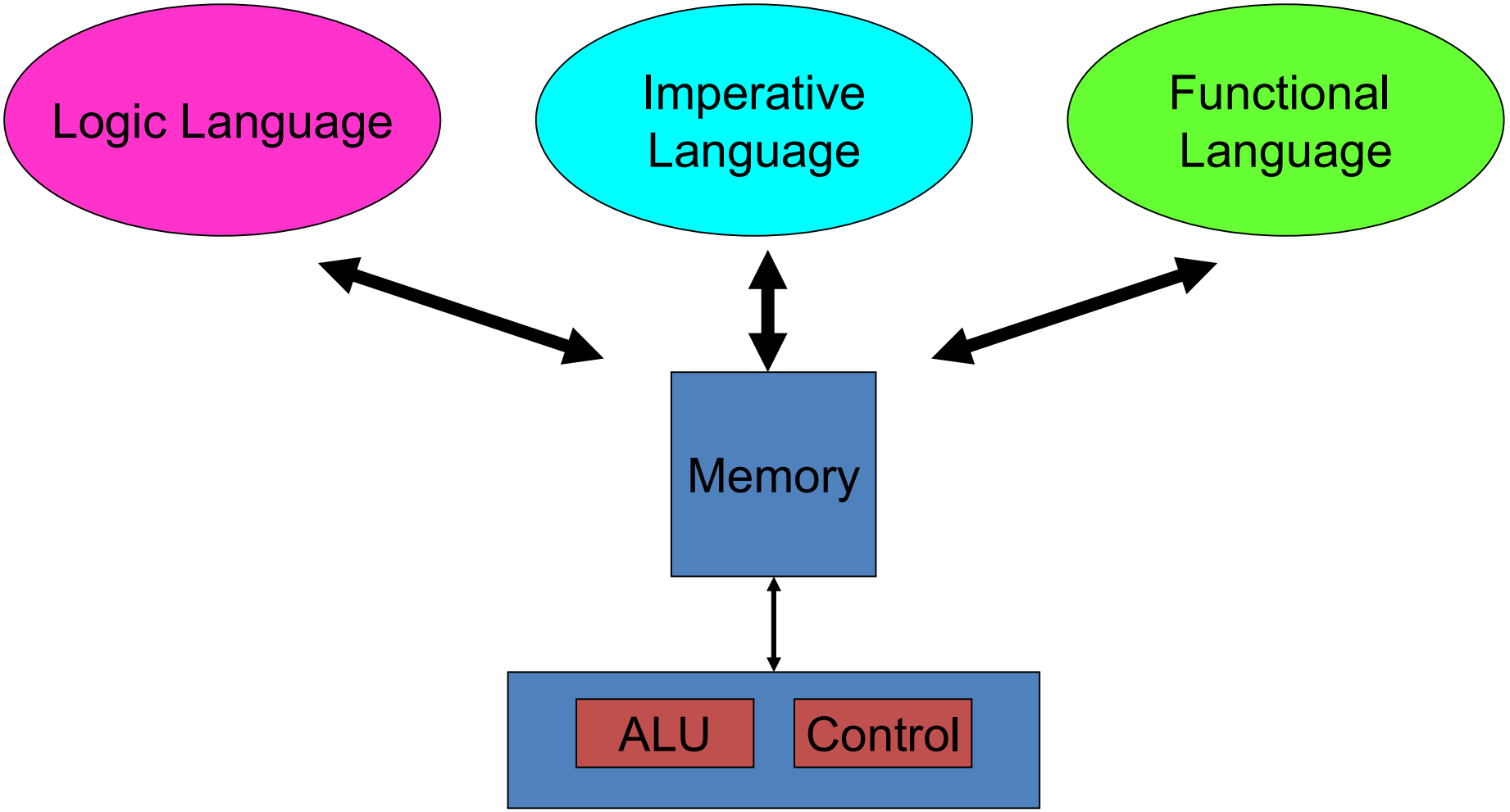
3) Logic

- Non-procedural
- Only supply relevant facts (predicate calculus) and inference rules (resolutions)
- System then infer the truth of given queries/goals
- Highly inefficient, small application areas (database, AI)

Example: a factorial function in Prolog

```
fact(X,1)  :-  X =:= 1.
fact(X,Fact)  :-
  X > 1, NewX is X - 1,
  fact(NewX,NF),
  Fact is X * NF.
```

Summary: Language Categories



von Neumann Architecture

Summary: Language Categories

Imperative

- Variables, assignment statements, and iteration
- Include languages that support object-oriented programming, scripting languages, visual languages

  Ex.: C, Java, Perl, JavaScript,

Functional

- Computing by applying functions to given parameters

  Ex.: LISP, Scheme, ML

Logic

- Rule-based (rules are specified in no particular order)

  Ex.: Prolog

Language evaluation criteria

- **Readability**: the ease with which programs can be read and understood
- **Writability**: the ease with which a language can be used to create programs
- **Reliability**: a program performs to its specifications under all conditions
- **Cost: total cost**

Evaluation Criteria: Readability

Overall simplicity

- A manageable set of features and constructs
- Minimal feature multiplicity
- Minimal operator overloading

Orthogonality

- A relatively small set of primitive constructs can be combined in a relatively small number of ways
- Every possible combination is legal

Data types

- Adequate predefined data types

Syntax considerations

- Identifier forms: flexible composition
- Special words and methods of forming compound statements
- Form and meaning: self-descriptive constructs, meaningful keywords

Evaluation Criteria: Writability

Simplicity and orthogonality

- Few constructs, a small number of primitives, a small set of rules for combining them

Support for abstraction

- The ability to define and use complex structures or operations in ways that allow details to be ignored

Expressivity

- A set of relatively convenient ways of specifying operations
- Strength and number of operators and predefined functions

Evaluation Criteria: Reliability

Type checking

- Testing for type errors

Exception handling

- Intercept run-time errors and take corrective measures

Aliasing

- Presence of two or more distinct referencing methods for the same memory location

Readability and writability

- A language that does not support “natural” ways of expressing an algorithm will require the use of “unnatural” approaches, and hence reduced reliability

Evaluation Criteria: Cost

- Training programmers to use the language
- Writing programs (closeness to particular applications)
- Compiling programs
- Executing programs
- Language implementation system: availability of free compilers
- Reliability: poor reliability leads to high costs
- Maintaining programs

Evaluation Criteria: Others

Portability

- The ease with which programs can be moved from one implementation to another

Generality

- The applicability to a wide range of applications

Well-definedness

- The completeness and precision of the language's official definition

Implementation Methods

Compilation

Programs are translated into machine language; includes JIT systems
Use: Large commercial applications

Pure Interpretation

Programs are interpreted by another program known as an interpreter
Use: Small programs or when efficiency is not an issue

Hybrid Implementation Systems

A compromise between compilers and pure interpreters
Use: Small and medium systems when efficiency is not the first concern

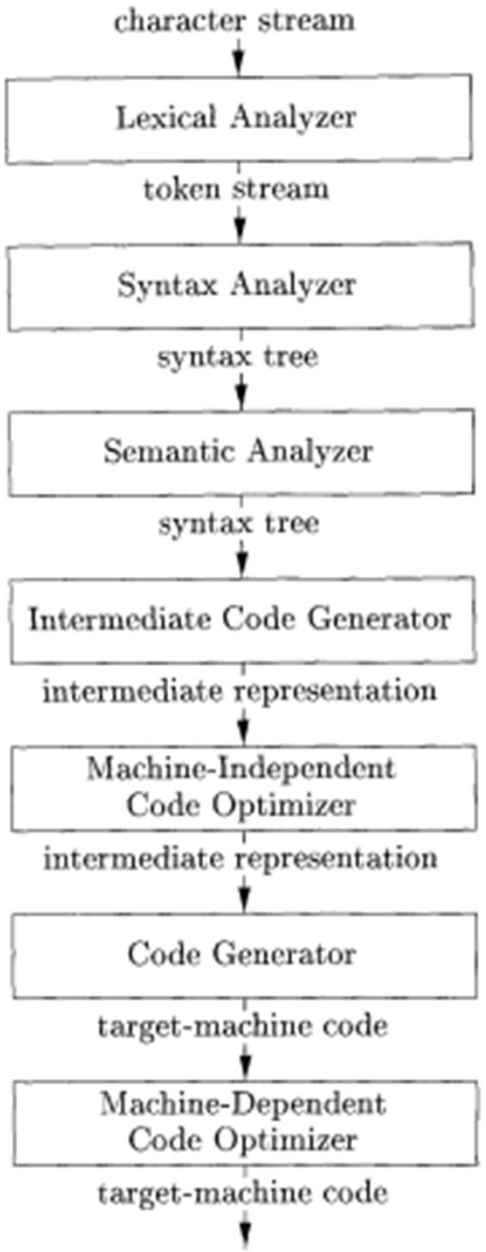
Compilation

Translate high-level program (source language) into machine code (machine language)
Slow translation, fast execution

Compilation process has several phases:

- lexical analysis: converts characters in the source program into lexical units
- syntax analysis: transforms lexical units into *parse trees* which represent the syntactic structure of program
- Semantics analysis: generate intermediate code
- code generation: machine code is generated

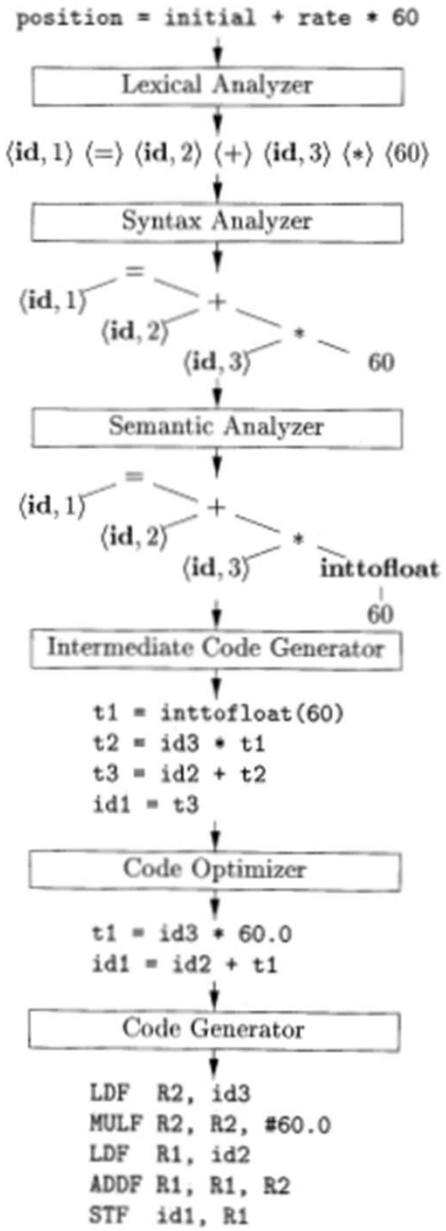
The Compilation Process



The Compilation Process: An Example

1	position	···
2	initial	···
3	rate	···

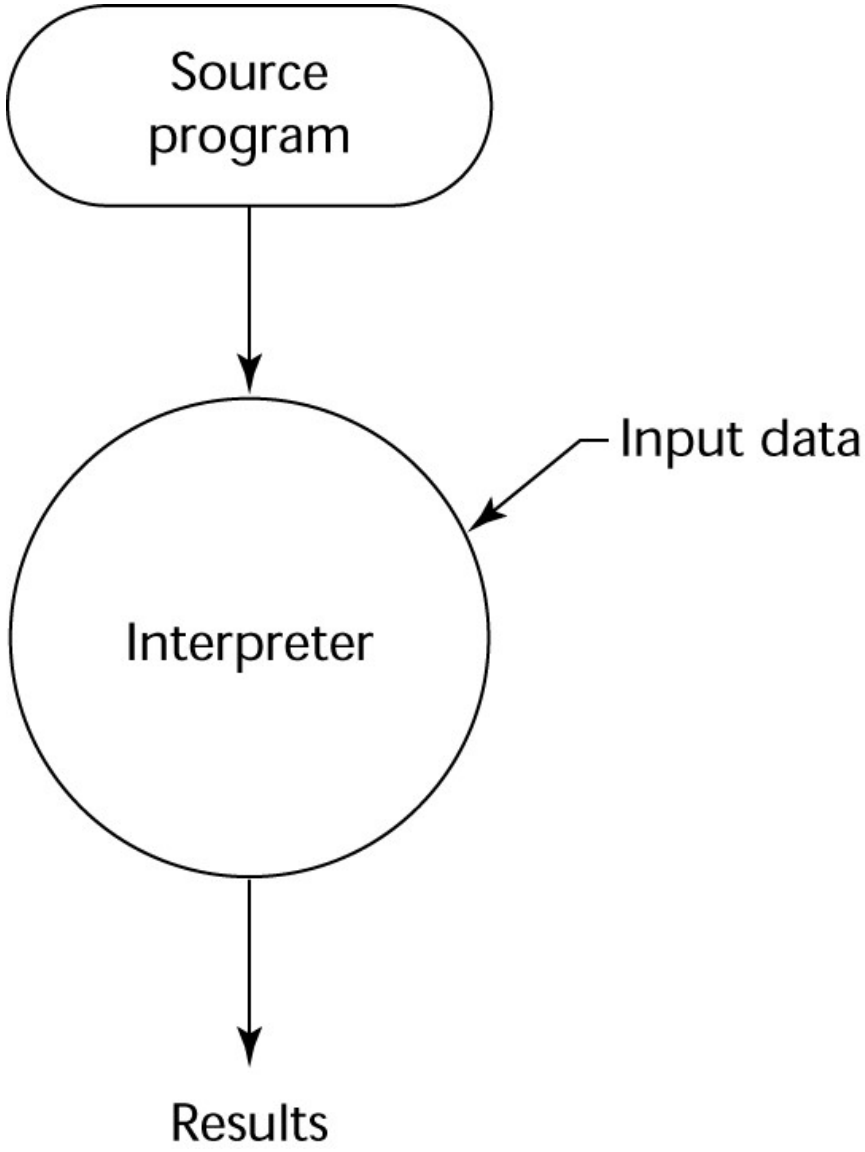
SYMBOL TABLE



Pure Interpretation

- No translation
- Easier implementation of programs (run-time errors can easily and immediately be displayed)
- Slower execution (10 to 100 times slower than compiled programs)
- Often requires more space
- Now rare for traditional high-level languages
- Significant comeback with some Web scripting languages (e.g., JavaScript, PHP)

Pure Interpretation Process



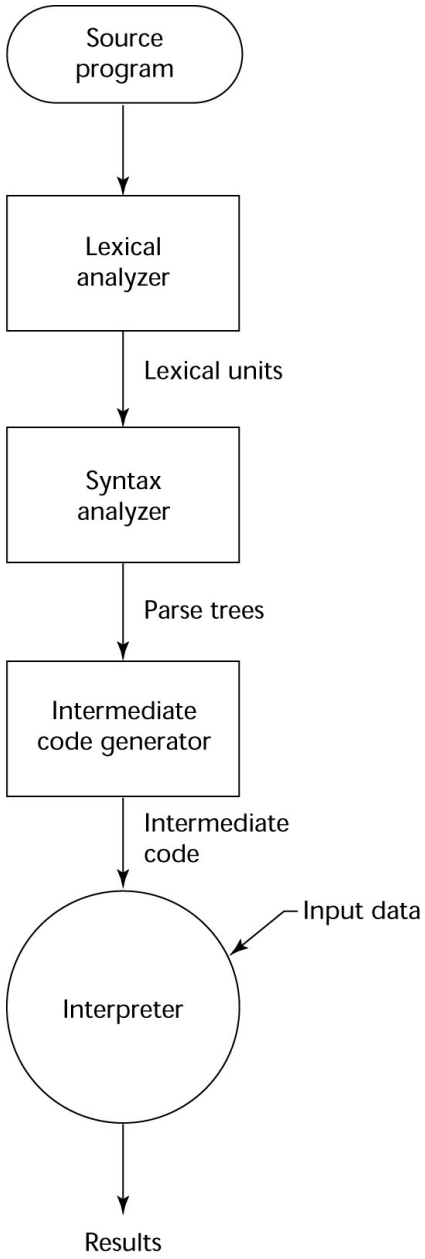
Hybrid Implementation Systems

- A compromise between compilers and pure interpreters
- A high-level language program is translated to an intermediate language that allows easy interpretation
- Faster than pure interpretation

Examples

- Perl programs are partially compiled to detect errors before interpretation
- Initial implementations of Java were hybrid; the intermediate form, *byte code*, provides portability to any machine that has a byte code interpreter and a run-time system (together, these are called *Java Virtual Machine*)

Hybrid Implementation Process



Just-in-Time Implementation Systems

- Initially translate programs to an intermediate language
- Then compile the intermediate language of the subprograms into machine code when they are called
- Machine code version is kept for subsequent calls
- JIT systems are widely used for Java programs
- .NET languages are implemented with a JIT system
- In essence, JIT systems are delayed compilers

Preprocessors

- Preprocessor macros (instructions) are commonly used to specify that code from another file is to be included
- A preprocessor processes a program immediately before the program is compiled to expand embedded preprocessor macros
- A well-known example: C preprocessor
- expands #include, #define, and similar macros

Books

Text Books:

1. Concepts of Programming Languages, Robert W. Sebesta, Pearson.
2. Programming Languages, Principles & Paradigms, Allen B Tucker, Robert E Noonan
3. Online resources

Syllabus

Component	Unit	Topics for Coverage	Chapter No.(Optional)
Component 1	Unit 1	Rationale for studying programming languages, criteria used for evaluating programming languages and language constructs, context free grammar, BNF, attribute grammars, semantics: operational, denotational, and axiomatic semantics, various phases of compilers. Design issues for: variables, data types.	1,3,5,6
	Unit 2	Design issues for expressions and assignment statements, control statements, subprograms and their implementation.	7, 8,9,10
Component 2	Unit 3	Data abstraction in-depth discussion of language features that support object-oriented programming (inheritance and dynamic method binding), and exception handling along with a brief discussion of event handling.	11,12,14
	Unit 4	Concurrency, programming paradigms: functional programming with Scheme, brief introductions to ML, Haskell, and F#. introduction to logic programming using Prolog.	13,15,16

THANK YOU