CSCI 305

Introduction

Reasons for Studying Concepts of PLs

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

Programing Domains

- Scientific Applications
- Business Applications
- Artificial Intelligence
- Systems Programming
- Web Software
- Entertainment

Language Evaluation Criteria

Characteristic	Readability	Writability	Reliability
Simplicity	X	X	X
Orthogonality	X	X	X
Control Structures	X	X	X
Data Types & Structures	X	X	X
Syntax Design	X	X	X
Support for Abstraction		X	X
Expressivity		X	X
Exception Handling			X
Restricted Aliasing			X

Language Evaluation Criteria

Characteristic????	Readability	Writability	Reliability
OO Support	?	?	?
Support for Interfaces		X	?
Support for Reflection		X	?
Portability	*	*	*
Tools Available	*	*	*
Unit Testability	*	*	*
Separation of Concerns	*	*	*
Market Value	*	*	*

Language Evaluation

CRITERIA

Readability

- Ease at which it can be understood
- Made popular by the introduction of software life cycle (70s)
- Machine vs. Human Orientation
- Must be evaluated in context of problem domain

Writability

- How easy programs can be created for a problem domain
- Writability typically has a subset of readability characteristics
- Like readability domain is key

Reliability

- A program is reliable if it preforms to specifications under all conditions
- Reliability typically isn’t considered per problem domain.
- Can be affected by factors outside of the programing language itself, consider program reliability vs. platform reliability

Of languages that allow us to evaluate Criteria

CHARACTERISTICS

Overall Simplicity

- Small vs. large number of basic constructs
- Most directly affect readability

Overall Simplicity (cont)

- Feature Multiplicity – two or more ways to accomplish a single operation

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

Overall Simplicity (cont)

- Operator Overloading

  int = int + int

  float = float + float

  struct = struct + struct

  array = array + array

  int = array + array

Overall Simplicity (end)

- Consider assembly language as the other extreme. Extremely simple, extremely hard to read.
- Programmers often use an learn a small number of the constructs if the set is large
- Readability issues arise between readers/writers of the same program.
- Simplicity is usually lightly considered in place of expectations for good design and programming practices

Orthogonality

- A relatively small set of primitive constructs can be combined in a relatively small number of ways
- AND every possible combination of primitive is legal and meaningful
- Meaning of an orthogonal language feature is independent of the context of appearance in the program (consider C increments)

Control Statements

- For, While, Loop, etc vs goto
- Not widely available until the 70s

```
loop1:
        if (incr >= 2) go to out;
loop2:
        if (sum > 100) go to next;
        sum += incr;
        go to loop2;
next:
        incr++;
        go to loop1;
out:
```

Data Type and Structures

- Consider Boolean

```
intVar = 1     vs
realVar = true
```

- Consider it's affect on orthogonality

C

```
if (intVar != 0)   or  if  (intVar == 1)
if (var)
```

Vs (C#, Java, etc)

```
if (realVar)            //  if(intVar) does not compile
```

Data Types and Structures (cont)

- Language C alternatives

```
typedef char bool;
bool = ‘y’;
bool = ‘n’;
If (bool == ‘y’)
```

Data Types and Structures (cont)

- Language C alternatives

```
#define true 1
#define false 0
typedef char bool;

bool = true;
if (bool == true)…
if (bool)…

if (bool != ‘z’)   // ??
```

Data Types and Structures (end)

- Language C alternatives

```
#if (__BORLANDC__ <= 0x460) || !defined(__cplusplus)
        typedef enum { false, true } bool;
#endif
```

- No longer portable (stdbool.h) now breaks in macros
- Same casting issues as before

Syntax Design

- Mainly affects readability

- Identifier forms

  boolean, bool, b

  integer, int, i

Syntax Design (cont)

- Special Words

  while, class, for, loop, struct

- Terminating Special contructs

  }

  end while (endwhile)

  end if (endif)

- Simplicity in reading vs writing
  - Reading: more reserved works
  - Writing: consistent and simple reserved words

Syntax Design (cont)

- Reserved words
  - Can reserved words be used as variable names? (Fortran 95 allows)

```
int if ;
int for = 2
int break = 3

for (if = 1; if < break; if++)
        if (if > for)
                break;
```

Syntax Design (cont)

- Form and meaning. It is helpful when statements/constructs match their meaning.
  - do -> while
  - static in C?
  - Unix commands?!?!

Support for Abstraction

- The ability to define and then use complicated structures or operations in ways that allow many of the details to be ignored
- Process Abstraction
- Data Abstraction
- Difference?

Expressivity

- The language provides very powerful operators that allow much computation with a small program (number of lines)
- Or – A language has convenient, rather than cumbersome, way of specifying computations.

    count++

    loop, while, do, for, foreach

Type Checking

- Testing for type errors in a given program, either by compiler or during program execution
- Compile type checking is less expensive – both for program efficiency and maintenance.

```
bool var1;
int var2 = var1;
```

```
>> Error: Cannot cast var1 (bool) to type int.
```

Type Checking (end)

```
function myFund(int value)
{
        return value;
}

bool var1;
myFunc(var1);
```

>> Error: MyFunc expected type (int), found type (bool).

Exception Handling

- The ability for a program to intercept run-time errors, take corrective measures, and then continue
- Widely available in Ada, C++, Java, C#. Virtually non-existent in many other languages.

Exception Handling (end)

```
public static void main(String[] args) throws Exception{
    try
    {
            int a,b;
            BufferedReader in =
                    new BufferedReader(new InputStreamReader(System.in));
            a = Integer.parseInt(in.readLine());
            b = Integer.parseInt(in.readLine());
    }
    catch(NumberFormatException ex)
    {
          System.out.println(ex.getMessage()   + " is not a numeric value.");
          System.exit(0);
    }
}
```

Aliasing

- Having two or more distinct names that can be used to access the same memory cell.
- Restricted Aliasing?

When criteria conflict

DESIGN TRADEOFFS

Language Design Trade-Offs

- Reliability vs. cost of execution
  - Example: Java demands all references to array elements be checked for proper indexing, which leads to increased execution costs

- Readability vs. writability

  Example: APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability

- Writability (flexibility) vs. reliability
  - Example: C/C++ pointers are powerful and very flexible but are unreliable

Characteristic	Readability	Writability	Reliability
Simplicity	X	X	X
Orthogonality	X	X	X
Control Structures	X	X	X
Data Types	X	X	X
Syntax Design	X	X	X
Support for Abstraction		X	X
Expressivity			
Exception Handling			
Restricted Aliasing			

Characteristic????	Readability	Writability	Reliability
OO Support	?	?	?
Support for Interfaces		X	?
Support for Reflection		X	?
Portability	*	*	*
Tools Available	*	*	*
Unit Testability	*	*	*
Separation of Concerns	*	*	*
Market Value	*	*	*

All things considered…

COST

Cost

- Cost of Training
- Cost to Write
- Cost of Compilation
- Cost of Execution (Optimization)
- Cost of System
- Cost of Reliability (poor)
- Cost of Maintenance
- Opportunity Cost

Computer Architecture and Programming Methodologies

INFLUENCES ON LANGUAGE DESIGN

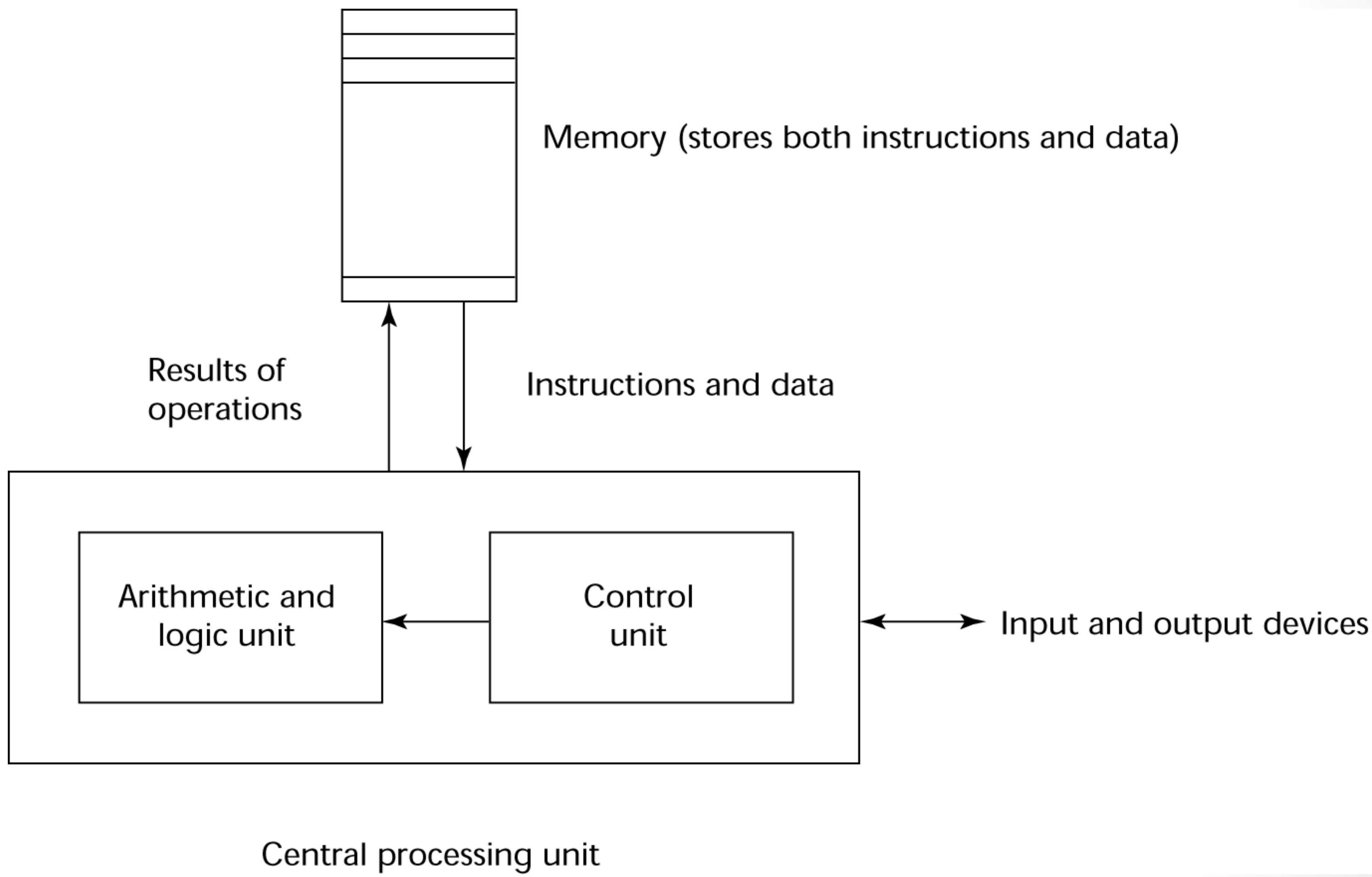
Influences on Language Design

- Computer Architecture
  - Languages are developed around the prevalent computer architecture, known as the *von Neumann* architecture
- Program Design Methodologies
  - New software development methodologies (e.g., object-oriented software development) led to new programming paradigms and by extension, new programming languages

Computer Architecture Influence

- Well-known computer architecture: Von Neumann
- Imperative languages, most dominant, because of von Neumann computers
  - Data and programs stored in 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

The von Neumann Architecture



The von Neumann Architecture

- Fetch-execute-cycle (on a von Neumann architecture computer)

```
initialize the program counter
repeat forever
  fetch the instruction pointed by the counter
  increment the counter
  decode the instruction
  execute the instruction
end repeat
```

Programming Methodologies Influences

- 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: Object-oriented programming
  - Data abstraction + inheritance + polymorphism

Imperative, Functional, Logic, Hybrid

LANGUAGE CATEGORIES

Language Categories

- Imperative
  - Central features are variables, assignment statements, and iteration
  - Include languages that support object-oriented programming
  - Include scripting languages
  - Include the visual languages
  - Examples: C, Java, Perl, JavaScript, Visual BASIC .NET, C++
- Functional
  - Main means of making computations is by applying functions to given parameters
  - Examples: LISP, Scheme, ML, F#
- Logic
  - Rule-based (rules are specified in no particular order)
  - Example: Prolog
- Markup/programming hybrid
  - Markup languages extended to support some programming
  - Examples: HTML, XML, XAML, JSTL, XSLT

Compilation, Interpretation, Hybrid, (JIT & Preprocessing)

IMPLEMENTATION METHODS

Implementation Methods

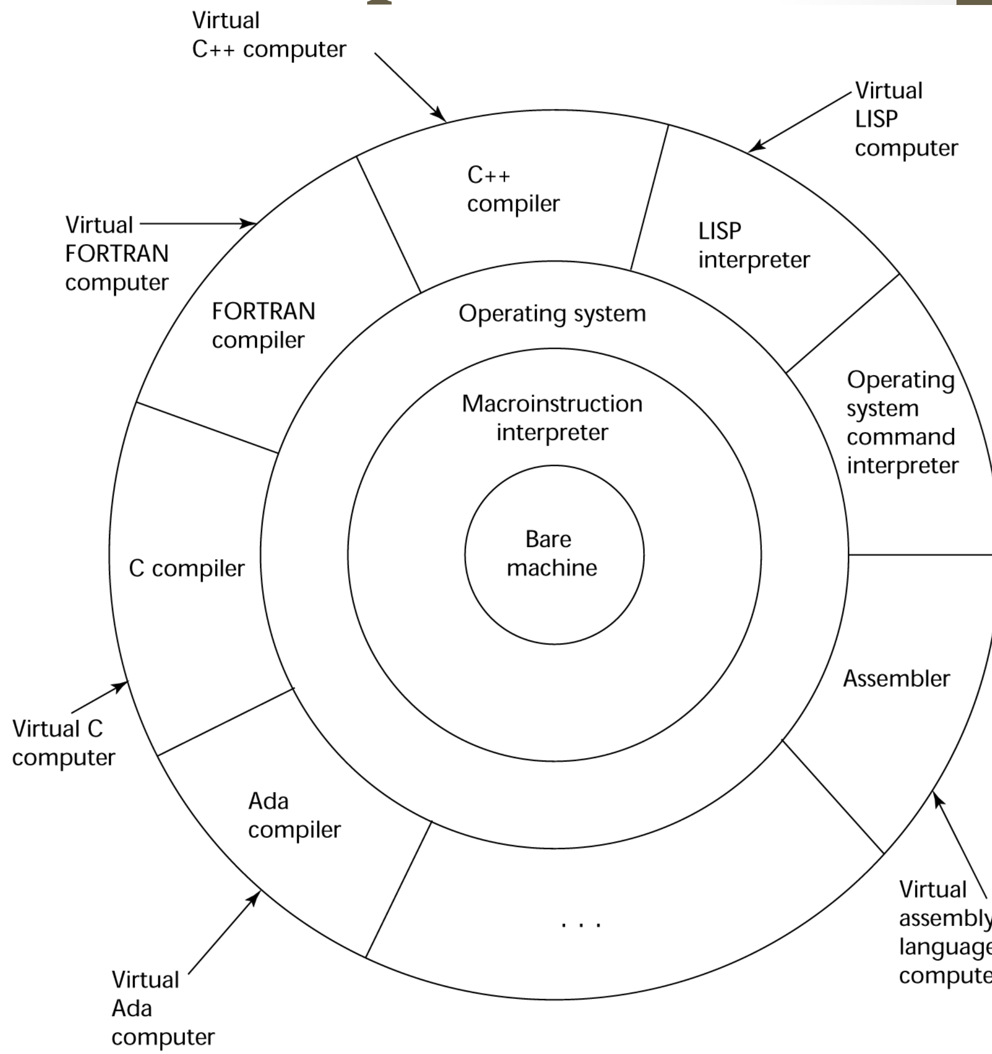
- Compilation
  - Programs are translated into machine language; includes JIT systems
  - Use: Large commercial applications

- Pure Interpretation
  - Programs are interpreted by another program known as an interpreter
  - Use: Small programs or when efficiency is not an issue / commercial web applications with caveats

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

Layered View of Computer

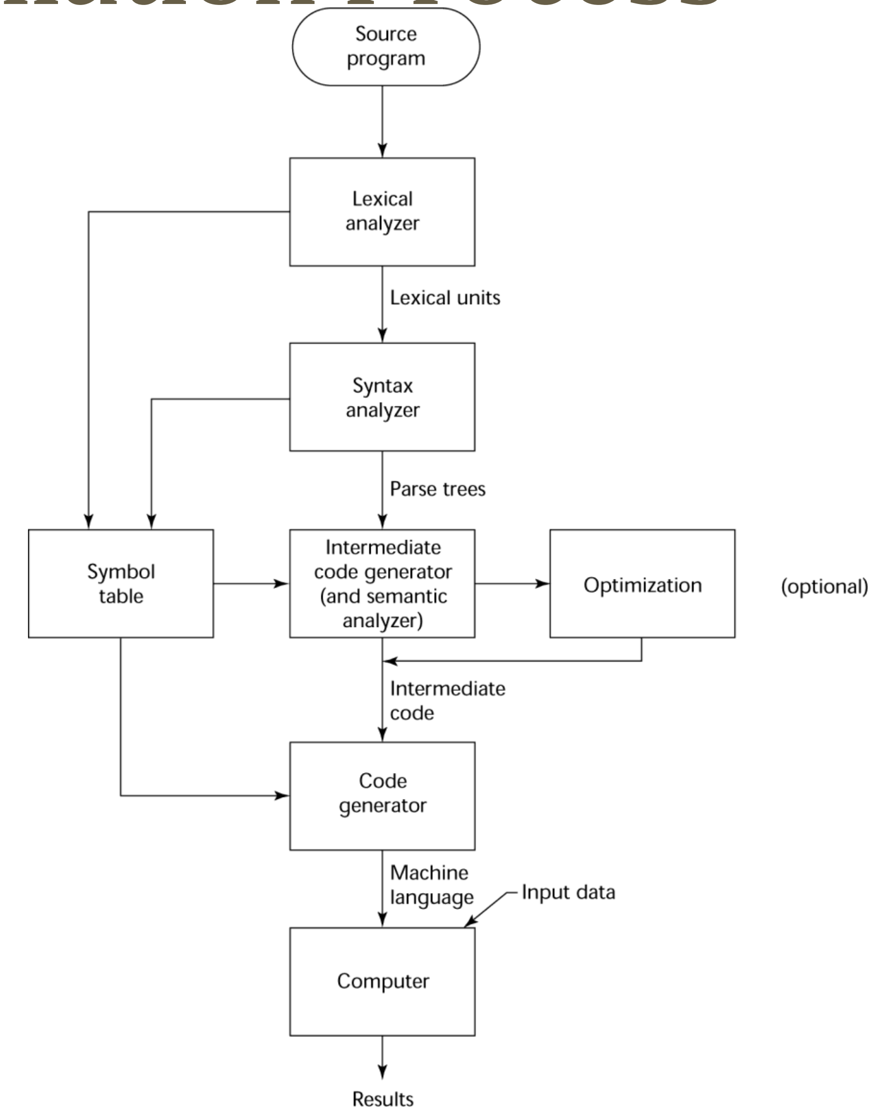
The operating system and language implementation are layered over machine interface of a computer



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



Additional Compilation Terminologies

- **Load module** (executable image): the user and system code together
- **Linking and loading**: the process of collecting system program units and linking them to a user program

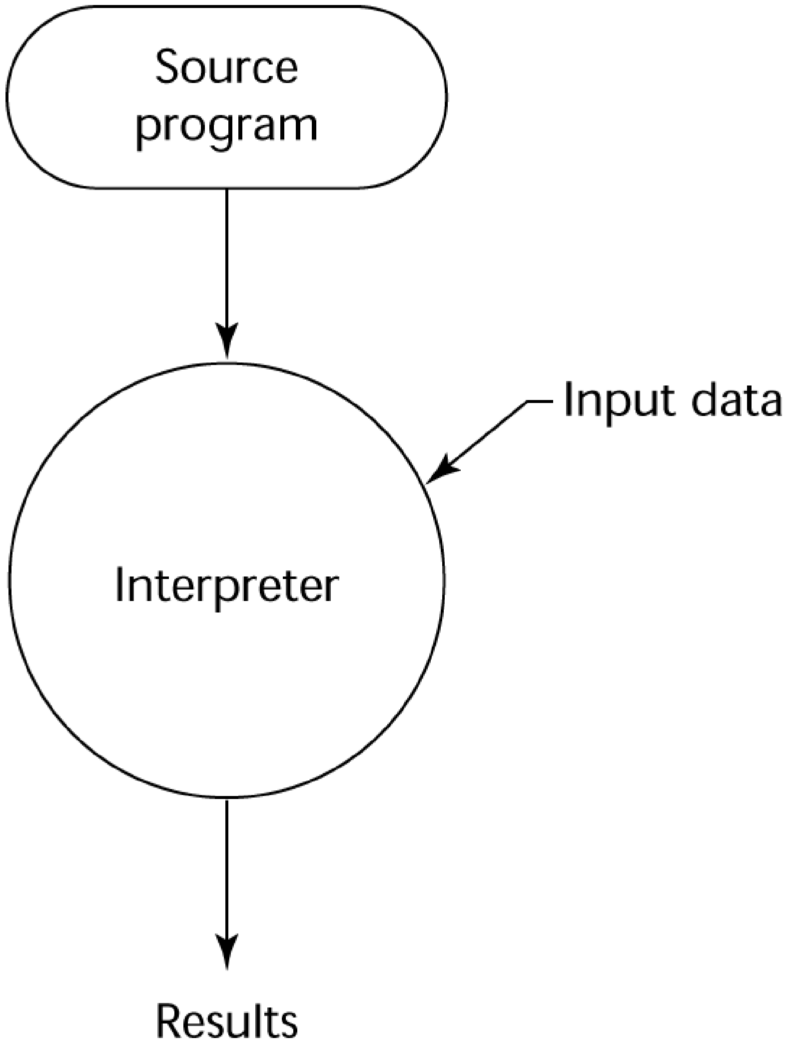
Von Neumann Bottleneck

- Connection speed between a computer's memory and its processor determines the speed of a computer
- Program instructions often can be executed much faster than the speed of the connection; the connection speed thus results in a *bottleneck*
- Known as the *von Neumann bottleneck*; it is the primary limiting factor in the speed of computers

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
- No compilation . No optimization. Bottleneck is in decoding rather than been processor and memory.
- Now rare for traditional high-level languages
- Significant comeback with some Web scripting languages (e.g., JavaScript, PHP)

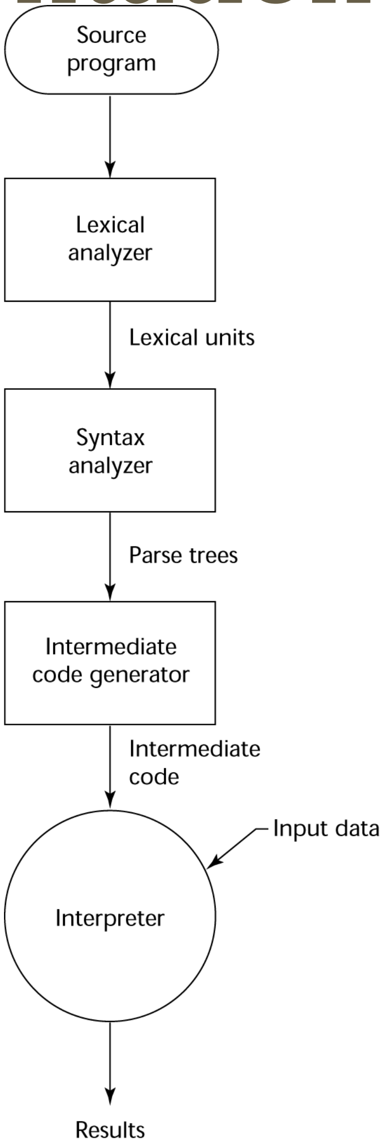
Pure Interpretation Process



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
- Purpose?
  - Allows code to be portable just through an interpreter
  - Allows some special machine time optimizations to be made
  - Can allow program to start faster. Program is brought into memory only as it's used.
  - Many arguments for against JIT caching/optimization (disk IO speed vs. JIT cost, etc).

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

Concluding Chapter 1

SUMMARY

Summary

- The study of programming languages is valuable for a number of reasons:
  - Increase our capacity to use different constructs
  - Enable us to choose languages more intelligently
  - Makes learning new languages easier
- Most important criteria for evaluating programming languages include:
  - Readability, writability, reliability, cost
- Major influences on language design have been machine architecture and software development methodologies
- The major methods of implementing programming languages are: compilation, pure interpretation, and hybrid implementation
- Final decision may always be trumped by platform viability in target market space.