

# CSCI: 4500/6500 Programming Languages

## Motivation & Big Picture

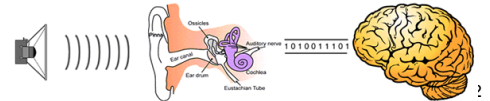


Maria Hybinette, UGA

1

# Why learn new [programming] languages?

- Communicate ideas better
  - » Become a better communicator (programming skills)
    - Translate ideas to words
  - » Become a better listener ('*compile*' information efficiency)
    - Translate words to ideas
- Comprehension:
  - » Speakers ability to translate ideas into language
  - » Listeners ability to translate work into ideas



Maria Hybinette, UGA

2

# Why study programming language concepts?

- One School of thought in Linguists:
  - » A Language "*shapes the way we think*" and determines "*what we can think about*" [Whorf-Sapir Hypothesis 1956]
  - » Programmers only skilled in one language may not have a deep understanding of concepts of other languages, whereas those who are multi-lingual can solve problems in many different ways.
- Help you choose appropriate languages for different application domains
- Increased ability to learn new languages
  - » Concepts have more similarities
- Easier to express ideas
- Helps you make better use of whatever language you do use

Maria Hybinette, UGA

3

# What is programming language?

- Translator between you (ideas), the programmer and the computer's native language
- Computer's native language:
  - » A computer is composed of on/off switches that tells the computer what to do.
    - 01111011 01111011 01111011
- How?
  - » Read by **assemblers**, **compilers** and **interpreters** and converted into machine code that the computer understands

Maria Hybinette, UGA

4

# What are components of a programming language?

- Like English -- each programming language has its own **grammar**, **syntax** (more details on this next week) and **semantics**.

Maria Hybinette, UGA

5

# Programming Language Definition

- Syntax
  - » Similar to the **grammar** of a natural language
  - » Most languages defined uses a **context free grammar** (Chomsky's type 2 grammar which can be described by non-deterministic PDA):
    - Production rules:  $A \rightarrow Y$ , where A is a *single* non terminal and Y is string of terminals and non terminals (regular languages are more restrictive  $Y \in \{\lambda, aA, a\}$ )
    - **Example**: the language of properly matched parenthesis is generated by the grammar:  $S \rightarrow |SS|(S)|\lambda$
    - $\langle \text{if-statement} \rangle ::= \text{if } \langle \text{expression} \rangle \langle \text{statement} \rangle [\text{else } \langle \text{statement} \rangle]$
- Semantics
  - » What does the program "mean"?
  - » Description of an if-statement [K&R 1988]:
    - An if-statement is executed by first evaluating its expression, which must have arithmetic or pointer type, including all side-effects, and if it compares unequal to 0, the statement following the expression is executed. If there is an else part, and the expression is 0, the statement following the else is executed.

Maria Hybinette, UGA

6

## Why are there so many programming languages?

- **Evolution:** We *learn 'better' ways* of doing things over time
- **Application Domains:** Different languages are good for different application domains with *different needs* that often conflict (next slide)
  - » **Special purpose:** Hardware and/or Software
- **Socio-Economical:** Proprietary interests, commercial advantage
- **Personal Preferences:** For example, some prefers recursive thinking and other prefers iterative thinking

Maria Hybinette, UGA

7

## Some Application Domains

- **Scientific computing:** Large number of floating point computations (e.g. Fortran)
- **Business applications:** Produce reports, use decimal numbers and characters (e.g. COBOL)
- **Artificial intelligence:** Symbols rather than numbers manipulated (e.g. LISP)
- **Systems programming:** Need efficiency because of continuous use, low-level access (e.g., C)
- **Web Software:** Eclectic collection of languages: markup (e.g., XHTML-- not a programming language), scripting (e.g., PHP), general-purpose (e.g., Java)
- **Academic:** Pascal, BASIC

Maria Hybinette, UGA

8

## What makes a language successful?

- **Expressiveness:** Easy to express things, easy use once fluent, "powerful" (C, Common Lisp, APL, Algol-68, Perl)
- **Learning curve:** Easy to learn (BASIC, Pascal, LOGO, Scheme)
- **Implementation:** Easy to implement (BASIC, Forth)
- **Efficient:** Possible to compile to very good (fast/small) code (Fortran)
- **Sponsorship:** Backing of a powerful sponsor (COBOL, PL/1, Ada, Visual Basic)
- **Cost:** Wide dissemination at minimal cost (Pascal, Turing, Java)

Maria Hybinette, UGA

9

## What makes a *good* language?

*No universal accepted metric for design.*

The "Art" of designing programming languages

Lets look at some **characteristics** and see how they affect the **criteria** below [Sebesta]:

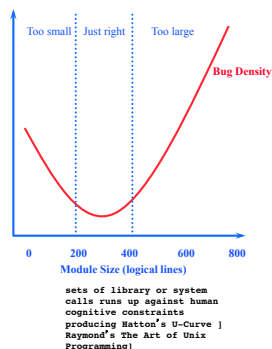
- **Readability:** the ease with which programs can be read and **understood**
- **Writability:** the ease with which a language can be used to **create** programs
- **Reliability:** **conformance** to specifications (i.e., performs to its specifications)
- **Cost:** the ultimate total cost (includes efficiency)

Maria Hybinette, UGA

10

## Characteristics

- **Simplicity:**
  - » Modularity, Compactness (encapsulation, abstraction)
  - » Orthogonality (mutually independent; well separated)
- **Expressivity**
- **Syntax**
- **Control Structures**
- **Data types & Structures**
- **Type checking**
- **Exception handling**



Maria Hybinette, UGA

11

## Compactness (Raymond)

- **Compact:** Fits inside a human head
  - » **Test:** Does an experienced user normally need a manual?
  - » Not the same as *weak* (can be powerful and flexible)
  - » Not the same as *easily learned*
    - Example: Lisp has a tricky model to learn then it becomes simple
  - » Not the same as *small* either (may be predictable and obvious to an experienced user with many pieces)
- **Semi-compact:** Need a reference or cheat sheet card

Maria Hybinette, UGA

12

## Compactness

- **The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information [Miller 1956]**
  - » Does a programmer have to remember more than **seven** entry points? Anything larger than this is unlikely to be strictly compact.
- **C & Python** are semi-compact
- **Perl, Java and shells** are not (especially since serious shell programming requires you to know half-a-dozen other tools like `sed(1)` and `awk(1)`).
- **C++** is anti-compact -- the language's designer has admitted that he doesn't expect any one programmer to ever understand it all.

Maria Hybinette, UGA

13

## Orthogonality

- Mathematically means: "Involving right angles"
- **Computing**: Operations/Instructions *do not have side effects*; each action changes just one thing without affecting others.
- **Small set of primitive constructs** can be combined in a relatively small number of ways (every possible combination is legal)
  - » Example monitor controls:
    - Brightness changed independently of the contrast level, color balance independently of both.
- **Don't repeat yourself rule**: Every piece of knowledge must have a *single*, unambiguous, authoritative representation within a system, or as Kernighan calls this: the Single Point Of Truth or **SPOT** rule.
- Easier to re-use mutually independent and well separated

Maria Hybinette, UGA

14

## Affects Readability

- **Overall simplicity**
  - » Compactness
  - » Few "feature multiplicity" (c+=1, c++)
    - (means of doing the same operation)
  - » Minimal operator overloading
- **Orthogonality**
- **Syntax considerations**
  - » Special words for compounds (e.g., `end if.`)
  - » Identifier forms (short forms of Fortran examples)
- **Control statements**
  - » Data structures facilities (true/1)
  - » Control structures (*while vs goto example next...*)

Simplicity	x
Control Structures	x
Data types & Structures	x
Syntax Design	x
Support Abstraction	
Expressivity	
Type Checking	
Exception Handling	
Restrictive Aliasing	

Maria Hybinette, UGA

15

## while vs goto

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

```
loop 1:
  if( incr >= 20 )
    goto out;
loop 2:
  if( sum > 100 )
    goto next;
  sum += incr;
  goto loop 2;
next:
  incr++;
  goto loop 1;
out:
```

- Comparison of a nested loop versus doing the same task in a language without adequate control statements.
- Which is more readable?

Maria Hybinette, UGA

16

## Affect 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
  - » Example: the inclusion of `for` statement in many modern languages

Simplicity	x
Orthogonality	
Control Structures	x
Data types & Structures	x
Syntax Design	x
Support Abstraction	x
Expressivity	x
Type Checking	
Exception Handling	
Restrictive Aliasing	

Maria Hybinette, UGA

17

## Affects 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 necessarily use "unnatural" approaches, and hence reduced reliability

Simplicity	x
Orthogonality	
Control Structures	x
Data types & Structures	x
Syntax Design	x
Support Abstraction	x
Expressivity	x
Type Checking	x
Exception Handling	x
Restrictive Aliasing	x

Maria Hybinette, UGA

18

## Summary

	Criteria		
	Readability	Writability	Reliability
Simplicity: Modular, Compact & Orthogonal	x	x	x
Control Structures	x	x	x
Data types & Structures	x	x	x
Syntax Design	x	x	x
Support Abstraction		x	x
Expressivity		x	x
Type Checking			x
Exception Handling			x
Restrictive Aliasing			x

Maria Hybinette, UGA

19

## Affects Cost

- Training programmers to use 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

Maria Hybinette, UGA

20

## 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

Maria Hybinette, UGA

21

## Design Trade-offs

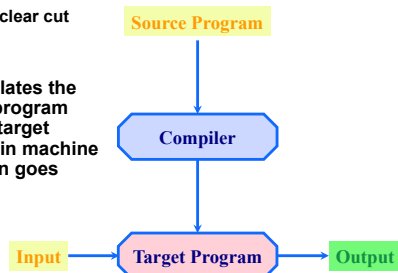
- **Reliability vs. cost of execution**
  - » Example: Java demands all references to array elements be checked for proper indexing but that leads to increased execution costs
- **Readability vs. writability**
  - » Example: APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability
- **Writability (flexibility) vs. reliability**
  - » Example: C++ pointers are powerful and very flexible but not reliably uses

Maria Hybinette, UGA

22

## Language Implementation Methods

- **Compilation vs. Interpretation**
  - » Not opposites, not a clear cut distinction
- **Pure Compilation**
  - » The compiler translates the high-level source program into an equivalent target program (typically in machine language), and then goes away:

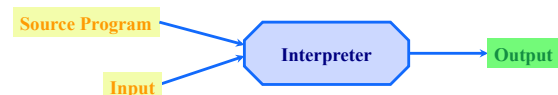


Maria Hybinette, UGA

23

## Compilation vs. Interpretation

- **Pure Interpretation**
  - » Interpreter stays around for the execution of the program
  - » Interpreter is the locus of control during execution



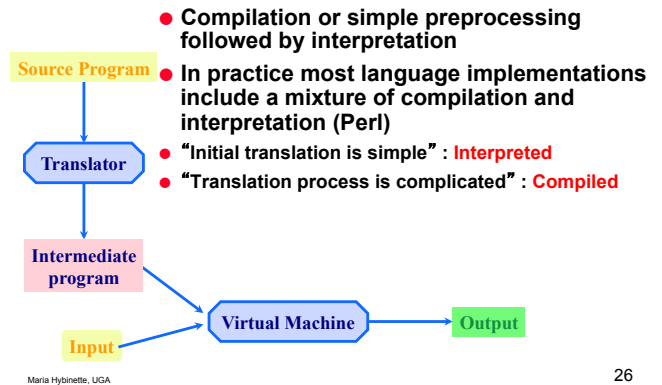
Maria Hybinette, UGA

24

# Compilation vs. Interpretation

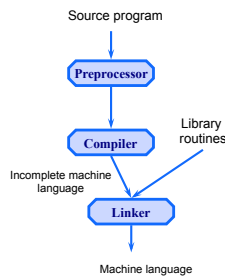
- **Interpretation:**
  - » Greater flexibility
  - » Better diagnostics (error messages, related to the text of source)
  - » Platform independence
  - » Example: Java, Perl, Ruby, Python, Lisp, Smalltalk
- **Compilation:**
  - » Better performance
  - » C, Fortran, Ada, Algol

# Hybrid: Compilation and Interpretation

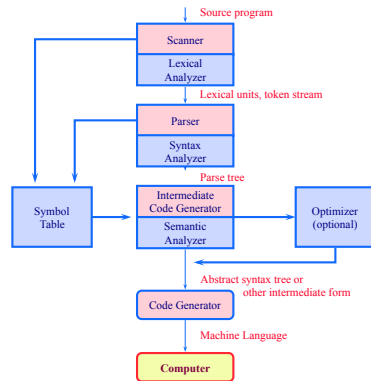


# Other implementation strategies

- **Preprocessor** - removes comments and white space, expand macros.
- **Library routines and linking** - math routines, system programs (e.g., I/O)
- **Post-compilation assembly** - compiler compiles to assembly. Facilitates debugging & isolate debugger from changes in machine language (only assembler need to be changed)
- **Just-In-Time Compilation** - delay compilation until last possible moment
  - » Lisp, Prolog - compiles on fly
  - » Java's JIT - byte code → machine code
  - » C# → .NET Common Intermediate Language (CIL) → machine code

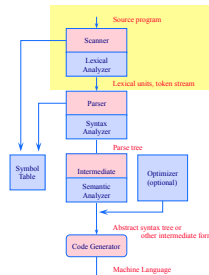


# Overview Compilation Process



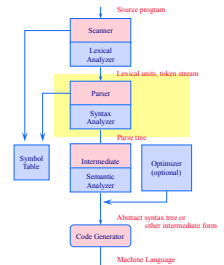
# Scanning

- Divides the program into "tokens"
  - » These are the smallest meaningful units; this saves time, since character-by-character processing is slow
    - you can design a parser to take characters instead of tokens as input, but it isn't pretty
- We can tune the scanner better if its job is simple; it also saves complexity (lots of it) for later stages
- Scanning is recognition of a **regular language**, e.g., via DFA
- Examples: Lex, Flex (tutorial next week)



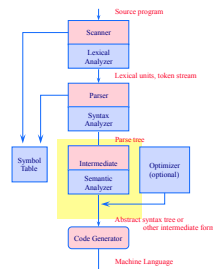
# Parsing

- A parser recognize how the tokens are **combined** in more complex **syntactic structures** determining its grammatical structure given a grammar.
- Informally, it finds the structure you can describe with syntax diagrams (the "circles and arrows" in a Pascal manual)
- Example Tools: Yacc, Bison (tutorial next week).



## Semantic Analysis

- Discovery of **meaning** in the program
- The compiler actually does what is called **STATIC** semantic analysis. That's the meaning that can be figured out at compile time
- Some things (e.g., array subscript out of bounds) can't be figured out until run time. Things like that are part of the program's **DYNAMIC** semantics

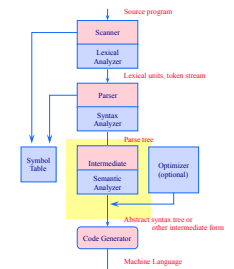


Maria Hybinette, UGA

31

## Intermediate Form (IF)

- Done after semantic analysis (if the program passes all checks)
- IFs are often chosen for machine independence, ease of optimization, or compactness (these are somewhat contradictory)
- They often resemble machine code for some imaginary idealized machine; e.g. a stack machine, or a machine with arbitrarily many registers
- Many compilers actually move the code through more than one IF

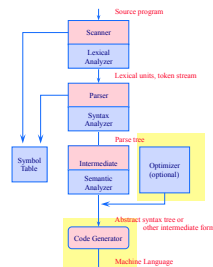


Maria Hybinette, UGA

32

## Optimization and Code Generation Phase

- **Optimization takes** an intermediate code program and produces another one that does the same thing faster, or in less space
  - » The term is a misnomer; we just improve code
  - » The optimization phase is optional
  - » Certain machine-specific optimizations (use of special instructions or addressing modes, etc.) may be performed during or after code generation
- **Code generation phase** produces assembly language or (sometime) relocatable machine language

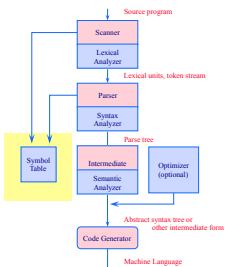


Maria Hybinette, UGA

33

## Symbol Table

- All phases rely on a symbol table that keeps track of all the identifiers in the program and what the compiler knows about them
- This symbol table may be retained (in some form) for use by a debugger, even after compilation has completed



Maria Hybinette, UGA

34

- Next week more details on syntax (tutorial)
- Next week:
  - » Programming language history
  - » Overview of different programming paradigms
    - Imperative, Functional, Logical, ...

Maria Hybinette, UGA

35