Early computers (1940s) cost millions of dollars and were programmed in machine language
– machine’s time more valuable than programmer’s
– machine language: bit sequences to perform low-level tasks; close to hardware
– tedious

Example: Euclid’s algorithm for GCD

```
pushl $ebp
movl $ebp, $esp
pushl $ebx
subl $4, $esp
addl $-16, $esp
call printf
movl $eax, $ebx
call printf
cmpl $eax, $ebx
je $:
addl $eax, $ebx
jmp $:
```

Less error-prone method needed
– assembly language: binary operations expressed with mnemonic abbreviations

Assembly language is specific to a certain machine, however
– tedious to re-write code for each computer type
– machine-independent language desired
– Fortran (mid-1950s) used a compiler to bridge the gap between high-level language and machine-dependent code
– many other languages followed

Why are there so many programming languages?
– evolution -- we've learned better ways of doing things over time
– socio-economic factors: proprietary interests, commercial advantage
– orientation toward special purposes
– orientation toward special hardware
– diverse ideas about what is pleasant to use
Introduction

• What makes a language successful?
  – easy to learn (BASIC, Pascal, LOGO, Scheme)
  – easy to express things, easy use once fluent, "powerful"
    (C, Common Lisp, APL, Algol-68, Perl)
  – easy to implement (BASIC, Forth)
  – possible to compile to very good (fast/small) code
    (Fortran)
  – backing of a powerful sponsor (COBOL, PL/1, Ada,
    Visual Basic)
  – wide dissemination at minimal cost (Pascal, Turing, Java)

Why study programming languages?

• Help you choose a language.
  – C vs. Modula-3 vs. C++ for systems programming
  – Fortran vs. APL vs. Ada for numerical computations
  – Ada vs. Modula-2 for embedded systems
  – Common Lisp vs. Scheme vs. ML for symbolic data manipulation
  – Java vs. C/CORBA for networked PC programs

Why study programming languages?

• Help you make better use of whatever language you use
  – understand obscure features:
    • In C, help you understand unions, arrays & pointers,
      separate compilation, varargs, catch and throw
    • In Common Lisp, help you understand first-class functions/closures, streams, catch and throw, symbol
      internals

Why study programming languages?

• Help you make better use of whatever language you use (2)
  – understand implementation costs: choose between alternative ways of doing things, based on knowledge of what
    will be done underneath:
    – use simple arithmetic equal (use x*x instead of x**2)
    – use C pointers or Pascal "with" statement to factor address calculations
    – avoid call by value with large data items in Pascal
    – avoid the use of call by name in Algol 60
    – choose between computation and table lookup (e.g. for cardinality operator in C or C++)
Why study programming languages?

- Help you make better use of whatever language you use (3)
  - figure out how to do things in languages that don't support them explicitly:
    - lack of suitable control structures in Fortran
    - use comments and programmer discipline for control structures
    - lack of recursion in Fortran, CSP, etc
    - write a recursive algorithm then use mechanical recursion elimination (even for things that aren't quite tail recursive)

- Help you make better use of whatever language you use (4)
  - figure out how to do things in languages that don't support them explicitly:
    - lack of named constants and enumerations in Fortran
    - use variables that are initialized once, then never changed
    - lack of modules in C and Pascal use comments and programmer discipline
    - lack of iterators in just about everything fake them with (member?) functions

Imperative languages

- Group languages as
  - imperative
    - von Neumann (Fortran, Pascal, Basic, C)
    - object-oriented (Smalltalk, Eiffel, C++)
    - scripting languages (Perl, Python, Java, JavaScript, PHP)
  - declarative
    - functional (Scheme, ML, pure Lisp, FP)
    - logic, constraint-based (Prolog, VisiCalc, RPG)

Imperative languages

- Imperative languages, particularly the von Neumann languages, predominate
  - They will occupy the bulk of our attention
- We also plan to spend a lot of time on functional, logic languages

Programming Language History

Compilation vs. Interpretation

- 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:
  
  ![Compilation vs. Interpretation Diagram](compilation_diagram.png)
Compilation vs. Interpretation

• Pure Interpretation
  – Interpreter stays around for the execution of the program
  – Interpreter is the locus of control during execution

Compilation vs. Interpretation

• Interpretation:
  – Greater flexibility
  – Better diagnostics (error messages)

• Compilation
  – Better performance

Compilation vs. Interpretation

• Common case is compilation or simple pre-processing, followed by interpretation
• Most language implementations include a mixture of both compilation and interpretation

Compilation vs. Interpretation

• Note that compilation does NOT have to produce machine language for some sort of hardware
• Compilation is translation from one language into another, with full analysis of the meaning of the input
• Compilation entails semantic understanding of what is being processed; pre-processing does not
• A pre-processor will often let errors through. A compiler hides further steps; a pre-processor does not

Compilation vs. Interpretation

• Many compiled languages have interpreted pieces, e.g., formats in Fortran or C
• Most use “virtual instructions”
  – set operations in Pascal
  – string manipulation in Basic
• Some compilers produce nothing but virtual instructions, e.g., Pascal P-code, Java byte code, Microsoft COM+
### Compilation vs. Interpretation

**Implementation strategies:**
- **Library of Routines and Linking**
  - Compiler uses a *linker* program to merge the appropriate *library* of subroutines (e.g., math functions such as sin, cos, log, etc.) into the final program:

```
Fортен программа ➔ Compiler ➔ Incomplete machine language

Incomplete machine language ➔ Linker ➔ Machine language program
```

- **Post-compilation Assembly**
  - Facilitates debugging (assembly language easier for people to read)
  - Isolates the compiler from changes in the format of machine language files (only assembler must be changed, is shared by many compilers):

```
Source program ➔ Compiler ➔ Assembly language

Assembly language ➔ Assembler ➔ Machine language
```

- **The C Preprocessor (conditional compilation)**
  - Preprocessor deletes portions of code, which allows several versions of a program to be built from the same source:

```
Source program ➔ Preprocessor ➔ Modified source program

Modified source program ➔ Compiler ➔ Assembly language
```

- **Source-to-Source Translation (C++)**
  - C++ implementations based on the early AT&T compiler generated an intermediate program in C, instead of an assembly language:

```
Source program ➔ Preprocessor ➔ Modified source program

Modified source program ➔ C++ compiler ➔ C code

C code ➔ C compiler ➔ Assembly language
```

- **Bootstrapping**
  - Pascal compiler, in Pascal, that generates Pascal code that generates Pascal code running on the Pascal interpreter:

```
Pascal compiler, in Pascal, that generates Pascal code ➔ Pascal compiler, in Pascal, that generates Pascal code running on the Pascal interpreter ➔ Pascal compiler, in Pascal, that generates machine language

Pascal compiler, in machine language, that generates machine language ➔ Pascal compiler, in machine language, that generates machine language running on the machine language interpreter ➔ Pascal compiler, in machine language, that generates machine language
```
Compilation vs. Interpretation

• Implementation strategies:
  – Compilation of Interpreted Languages
    • The compiler generates code that makes assumptions about decisions that won’t be finalized until runtime. If these assumptions are valid, the code runs very fast. If not, a dynamic check will revert to the interpreter.

• Implementation strategies:
  – Dynamic and Just-in-Time Compilation
    • In some cases a programming system may deliberately delay compilation until the last possible moment.
      – Lisp or Prolog invoke the compiler on the fly, to translate newly created source into machine language, or to optimize the code for a particular input set.
      – The Java language definition defines a machine-independent intermediate form known as byte code. Byte code is the standard format for distribution of Java programs.
      – The main C# compiler produces .NET Common Intermediate Language (CIL), which is then translated into machine code immediately prior to execution.

• Implementation strategies:
  – Microcode
    • Assembly-level instruction set is not implemented in hardware; it runs on an interpreter.
    • Interpreter is written in low-level instructions (microcode or firmware), which are stored in read-only memory and executed by the hardware.

• Compilers exist for some interpreted languages, but they aren’t pure:
  – selective compilation of compilable pieces and extra-sophisticated pre-processing of remaining source.
  – Interpretation of parts of code, at least, is still necessary for reasons above.

• Unconventional compilers
  – text formatters
  – silicon compilers
  – query language processors

Programming Environment Tools

• Tools

<table>
<thead>
<tr>
<th>Type</th>
<th>Unix examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Editors</td>
<td>vi, emacs</td>
</tr>
<tr>
<td>Pretty printers</td>
<td>koi, pandoc</td>
</tr>
<tr>
<td>Preprocessors</td>
<td>cpp, sed, awk</td>
</tr>
<tr>
<td>Debuggers</td>
<td>gdb, dbx, gdb</td>
</tr>
<tr>
<td>Style checkers</td>
<td>lint, purify</td>
</tr>
<tr>
<td>Module management</td>
<td>make, cc</td>
</tr>
<tr>
<td>Version manage</td>
<td>gnu, ccs</td>
</tr>
<tr>
<td>Assemblers</td>
<td>das</td>
</tr>
<tr>
<td>Link editors</td>
<td>ld, ldd</td>
</tr>
<tr>
<td>Pattern tools</td>
<td>flex, lex, cc, gcc</td>
</tr>
<tr>
<td>Program cross-references</td>
<td>ctags</td>
</tr>
</tbody>
</table>

An Overview of Compilation

• Phases of Compilation

  - Character stream
  - Token stream
  - Parse tree
  - Syntax tree or other intermediate form
  - Machine-independent code improvement (optional)
  - Target code generation
  - Assemble or machine-language, or other target language
  - Machine-specific code improvement (optional)
  - Symbol table
An Overview of Compilation

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

- **Parsing** is recognition of a context-free language, e.g., via PDA
  - Parsing discovers the "context free" structure of the program
  - Informally, it finds the structure you can describe with syntax diagrams (the "circles and arrows" in a Pascal manual)

- **Semantic analysis** is the 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

- **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

- **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
- **Code generation phase** produces assembly language or (sometime) relocatable machine language

- **Certain machine-specific optimizations** (use of special instructions or addressing modes, etc.) may be performed during or after target code generation

- **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
An Overview of Compilation

• Lexical and Syntax Analysis
  – GCD Program (Pascal)
    
    program gcd(input, output);
    var i, j : integer;
    begin
      read(i, j);
      while i <> j do
        if i > j then i := i - j
        else j := j - i;
    writeln(i)
    end.

An Overview of Compilation

• Lexical and Syntax Analysis
  – GCD Program Tokens
    
    program gcd ( input , output ) ;
    var i , j : integer ;
    begin
      while i <> j do
        if i > j then i := i - j
        else j := j - i;
    writeln ( i )
    end .

An Overview of Compilation

• Lexical and Syntax Analysis
  – Context-Free Grammar and Parsing
    
    Parsing organizes tokens into a parse tree that represents higher-level constructs in terms of their constituents.
    Potentially recursive rules known as context-free grammar define the ways in which these constituents combine.

An Overview of Compilation

• Context-Free Grammar and Parsing
  – Example (Pascal program)
    
    program gcd ( input , output ) ;
    var i , j : integer ;
    begin
      while i <> j do
        if i > j then i := i - j
        else j := j - i;
    writeln ( i )
    end .

An Overview of Compilation

• Lexical and Syntax Analysis
  – GCD Program (Pascal)
    
    program gcd(input, output);
    var i, j : integer;
    begin
      read(i, j);
      while i <> j do
        if i > j then i := i - j
        else j := j - i;
    writeln(i)
    end.

An Overview of Compilation

• Lexical and Syntax Analysis
  – GCD Program Tokens
    
    program gcd ( input , output ) ;
    var i , j : integer ;
    begin
      while i <> j do
        if i > j then i := i - j
        else j := j - i;
    writeln ( i )
    end .

An Overview of Compilation

• Context-Free Grammar and Parsing
  – Example (Pascal program)
    
    program gcd ( input , output ) ;
    var i , j : integer ;
    begin
      while i <> j do
        if i > j then i := i - j
        else j := j - i;
    writeln ( i )
    end .

An Overview of Compilation

• Lexical and Syntax Analysis
  – GCD Program (Pascal)
    
    program gcd(input, output);
    var i, j : integer;
    begin
      read(i, j);
      while i <> j do
        if i > j then i := i - j
        else j := j - i;
    writeln(i)
    end.

An Overview of Compilation

• Lexical and Syntax Analysis
  – GCD Program Tokens
    
    program gcd ( input , output ) ;
    var i , j : integer ;
    begin
      while i <> j do
        if i > j then i := i - j
        else j := j - i;
    writeln ( i )
    end .

An Overview of Compilation

• Context-Free Grammar and Parsing
  – Example (Pascal program)
    
    program gcd ( input , output ) ;
    var i , j : integer ;
    begin
      while i <> j do
        if i > j then i := i - j
        else j := j - i;
    writeln ( i )
    end .

An Overview of Compilation

• Context-Free Grammar and Parsing
  – GCD Program Parse Tree
    
    ![GCD Program Parse Tree](image)

An Overview of Compilation

• Context-Free Grammar and Parsing
  – GCD Program Parse Tree (continued)
    
    ![GCD Program Parse Tree](image)
An Overview of Compilation

- Syntax Tree
  - GCD Program Parse Tree

Figure 1.1: Syntax tree and symbol table for the GCD program.