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

Less error-prone method needed

- assembly language: binary operations expressed with mnemonic abbreviations

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

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
Introduction

- What makes a language successful?
  - easy to learn (BASIC, Pascal, LOGO, Scheme)
  - easy to express things, easy to 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, C#)
  - wide dissemination at minimal cost (Pascal, Turing, Java)

Why do we have programming languages?

- What is a language for?
  - way of thinking -- way of expressing algorithms
  - languages from the user's point of view
  - abstraction of virtual machine -- way of specifying what you want the hardware to do without getting down into the bits
  - languages from the implementor's point of view

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 (1)
  - understand obscure features:
    - In C, help you understand unions, arrays, pointers, separate compilation
    - 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 $x^2$ instead of $x\times x$)
    - 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)

**Imperative languages**

- Group languages as
  - imperative
    - von Neumann (Fortran, Pascal, Basic, C)
    - object-oriented (Smalltalk, Eiffel, C++)
    - scripting languages (Perl, Python, 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**


**Programming Language Properties**

- Programming languages have four properties:
  - syntax
  - naming
  - types
  - semantics
Programming Language Properties

• Syntax
  – precise description of all grammatically correct programs of that language
  – answers questions such as
    • what are the basic statements for the language?
    • how do I write …?
    • why is this a syntax error?

• Naming
  – many entities in a program have names
    • variables, types
    • functions, parameters
    • classes, objects
  – named entities in a running program bound by
    • scope
    • visibility
    • type
    • lifetime

• Types
  – collection of values and collection of operations on those values
  – simple types: numbers, characters, booleans, …
  – structured types: strings, lists, trees, hash tables
  – complex types: functions, classes, …
  – a language’s type system helps to determine legal operations and to detect type errors

• Semantics
  – the meaning of a program
  – provides answers to questions
    • what does each statement mean?
    • what underlying model governs run-time behavior, such as function calls?
    • how are objects allocated to memory at run-time?
    • how do interpreters work in relation to semantics?

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:

Source program → Compiler → Target program
Input → Target program → Output

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

Source program → Interpreter → Output
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
• Some 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., print 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

• Many compilers are self-hosting
  – they are written in the language they compile
  – e.g., C compiler written in C
• how?
  – bootstrapping
  – write small interpreter
  – hand-translate small number of statements into assembly
  – extend through incremental runs of the compiler through itself

Compilation vs. Interpretation

• Implementation strategies:
  – Preprocessor
    • Removes comments and white space
    • Groups characters into tokens (keywords, identifiers, numbers, symbols)
    • Expands abbreviations in the style of a macro assembler
    • Identifies higher-level syntactic structures (loops, subroutines)
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:

  Fortran program $\rightarrow$ Compiler $\rightarrow$ Incomplete machine language

  Incomplete machine language $\rightarrow$ Linker $\rightarrow$ Machine language program

Compilation vs. Interpretation

- Implementation strategies:
  - 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 $\rightarrow$ Compiler $\rightarrow$ Assembly language

  Assembly language $\rightarrow$ Assembler $\rightarrow$ Machine language

Compilation vs. Interpretation

- Implementation strategies:
  - 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 $\rightarrow$ Preprocessor $\rightarrow$ Modified source program

  Modified source program $\rightarrow$ Compiler $\rightarrow$ Assembly language

Compilation vs. Interpretation

- Implementation strategies:
  - 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 $\rightarrow$ Preprocessor $\rightarrow$ Modified source program

  Modified source program $\rightarrow$ C++ compiler $\rightarrow$ C code

  C code $\rightarrow$ C compiler $\rightarrow$ Assembly language

Compilation vs. Interpretation

- Implementation strategies:
  - Bootstrapping

  Pascal compiler, in Pascal, that generates machine language $\rightarrow$ Pascal compiler, in Pascal, that generates P-code, running on the Pascal interpreter

  Pascal compiler, in P-code, that generates machine language $\rightarrow$ Pascal compiler, in P-code, that generates P-code, running on the P-code interpreter

  Pascal compiler, in P-code, that generates machine language, running on the P-code interpreter

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.
Compilation vs. Interpretation

• 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.
  – 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>cb, indent</td>
</tr>
<tr>
<td>Preprocessors (grep, etc.)</td>
<td>grep, re, tac, etc.</td>
</tr>
<tr>
<td>Debuggers</td>
<td>gdb, adb, dbx, db</td>
</tr>
<tr>
<td>Style checkers</td>
<td>lint, purify</td>
</tr>
<tr>
<td>Mobile managers</td>
<td>make</td>
</tr>
<tr>
<td>Virtual machines</td>
<td>vmc, xre</td>
</tr>
<tr>
<td>Assemblers</td>
<td>as</td>
</tr>
<tr>
<td>Link editors, loaders</td>
<td>ld, ld.so</td>
</tr>
<tr>
<td>Parallel tools</td>
<td>axe, lex, cc, cc</td>
</tr>
<tr>
<td>Program cross-compiler</td>
<td>Cc</td>
</tr>
</tbody>
</table>

An Overview of Compilation

• Phases 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
**An Overview of Compilation**

- **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

- **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.
    ```

- 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 Tokens
  • Scanning (lexical analysis) and parsing recognize the
    structure of the program, groups characters into
    tokens, the smallest meaningful units of the program

```plaintext
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

Context-Free Grammar and Parsing
- Example (Pascal program)

```plaintext
program — PROGRAM id ( id more_ids ) ; begin .
where:
  block —- labels constants types variables subroutines BEGIN stat
  and
  more_ids —- , id more_ids
  or
  more_ids —- e
```

An Overview of Compilation

Context-Free Grammar and Parsing
- GCD Program Parse Tree

An Overview of Compilation

Context-Free Grammar and Parsing
- GCD Program Parse Tree (continued)

An Overview of Compilation

Syntax Tree
- GCD Program Parse Tree

Figure 1.6 Syntax tree and symbol table for the GCD program.