

# Introduction

# Compilers are Translators

## Source Code

- ☐ C/C++
- ☐ Fortran
- ☐ Java
- ☐ PERL
- ☐ MATLAB
- ☐ Natural Language
- ☐ ...

translate



## Target Code

- ☐ Machine Code
- ☐ Transformed Code (C, Java, ...)
- ☐ Virtual Machine Code
- ☐ Lower Level Commands
- ☐ Semantic Components
- ☐ ...

# Translation Mechanisms

## ❏ Compilation

- To translate a source program in one language into an executable program in another language and produce results while executing the new program
- Examples: C, C++, FORTRAN

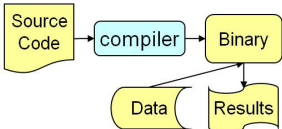
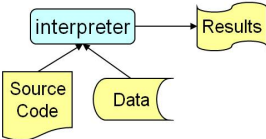
## ❏ Interpretation

- To read a source program and produce the results while understanding that program
- Examples: BASIC, LISP

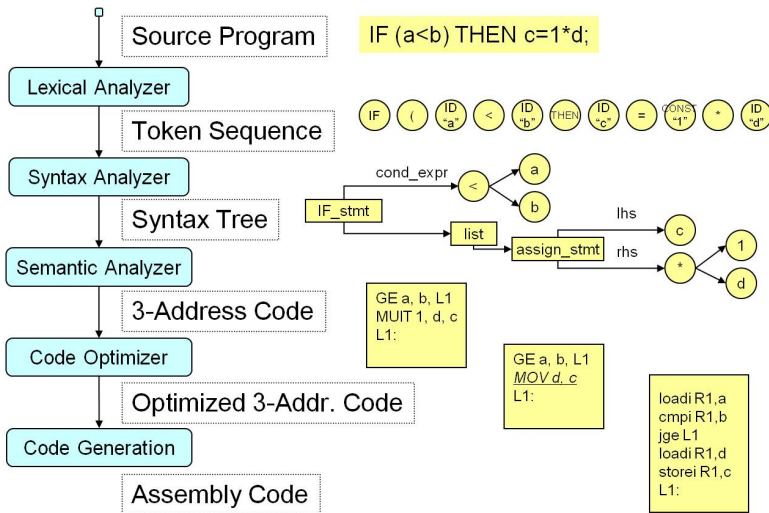
## ❏ Hybrid

- Case study: Java JVM
  - (1) Translate source code to bytecode
  - (2.a) Execute by interpretation on a JVM
  - or
  - (2.b) Execute by compilation using a JIT

# Comparison of Compiler/Interpreter

	Compiler	Interpreter
Overview	 <pre> graph LR     SC[Source Code] --&gt; C[compiler]     C --&gt; B[Binary]     B --&gt; D[Data]     B --&gt; R[Results]           </pre>	 <pre> graph LR     SC[Source Code] --&gt; I[interpreter]     D[Data] --&gt; I     I --&gt; R[Results]           </pre>
Advantages	Fast program execution; Exploit architecture features;	Machine independent; Easy to debug; Flexible to modify;
Disadvantages	Pre-processing of program; Complexity;	Execution overhead; Space overhead;

# Phases of a Modern Compiler



# Lexical and Syntax Analysis

## Lexical Analysis

- Recognize token ? smallest unit over letters
- Analyze input (strings of characters) from source
  - Scan from left to right
  - Report errors

## Syntax Analysis

- Group tokens into hierarchy groups
  - Differentiate if-statement, while-statement, ...
  - Report errors

# Semantic Analysis

- Determine the meaning using the structure
  - Checks are performed to ensure components fit together meaningfully
    - Information is added
    - Limited analysis to catch inconsistencies e.g. type checking
  - Put semantic meaningful items in the structure
    - Produce IR (intermediate representation)
    - Easier to generate machine code from IR

# Code Optimization and Generation

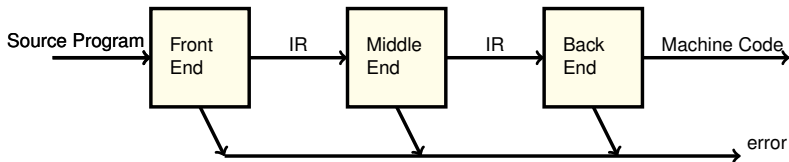
- ❏ Code optimization: modify program representation so that program
  - Run faster
  - Use less memory
  - In general, reduce the consumed resources
- ❏ Code generation: produce target code
  - Instruction selection
  - Memory allocation
  - Resource allocation — registers, processors, etc.



# Symbol Table Management

- ❑ Collect and maintain information about ids
  - Attributes: type, storage, scope, number, ...
- ❑ Used by most compiler passes
  - Phases to add information:  
lexical, parsing, semantic
  - Phases to use information:  
semantic, code optimization, code generation
- ❑ Debuggers use some form of symbol table
  - “gcc -g ...” keeps the symbol table in the object code

# Traditional Three-pass Compilation



## Code Optimization

- Analyzes IR and rewrites (or transforms) IR
- Primary goal is to reduce running time of the compiled code
  - May also improve space, power consumption, ...
- Must preserve “meaning” of the code
  - Measured by values of named variables

# Distinction Between Phases and Passes

- ❑ Passes: number of times through a program representation
  - 1-pass, 2-pass, multi-pass compilation
  - Language becomes more complex — more passes
- ❑ Phases: conceptual and sometimes physical stages
  - Symbol table coordinates information between phases
  - Phases are not completely separate
    - Semantic phase may do things that syntax phase should do
    - Interaction is possible

# Compiler Tools

## Automatic Generators

Lexical Analysis — Lex, Flex

Syntax Analysis — Yacc, Bison

Semantic Analysis

Code Optimization

Code Generation

# Compilers vs. Language Design

- ❑ There is a strong mutual influence
  - Hard to compile languages are hard to read
  - Easy to compile languages lead to quality compilers, better code, smaller compiler, more reliable, cheaper, wider use, better diagnostics
- ❑ Example: Dynamic Typing
  - seems convenient because type declaration is not needed

However,

- hard to read because the type of an identifier is not known
- hard to compile because the compiler cannot make assumptions about the identifier's type

# Example: Dynamic Typing

 Is the following program correct?

```
int x=100;
function f() {
    string x;
    ...
}
function g() {
    int * x;
    ...
}
main() {
    ...
    if (...)
        f();
    else
        g();
    x=x+1;
    strcpy(x, "test");
}
```

# Compilers vs. Computer Architecture Influence

- ❑ Complex instructions were available when programming at assembly level
  - CISC: Complex Instruction Set Computing
- ❑ RISC architecture became popular with the advent of high-level languages
  - RISC: Reduced Instruction Set Computing
  - Why is RISC preferred by a compiler?
- ❑ Today, the development of new instruction set architectures (ISA) is heavily influenced by compilation advances
  - MMX: Multimedia Instruction Set Extension
  - NP: Network Processors

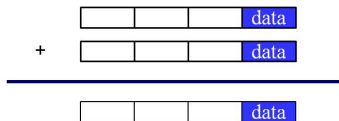
# Computer Architecture Influence

- ❏ Von Neumann Architecture
  - Well-known, widely used
- ❏ 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



# Example: MMX

- ❑ Multimedia applications need to process byte stream intensively
  - Spend most of its time in byte data computation
- ❑ RISC machine need to load the data to registers before computation
  - Register R1, R2, R3, ... are 32-bit or 64-bit registers



Normal ISA: higher order bits are wasted



MMX: need byte-level insulation