Introduction

Chapter 1

Slides adapted from :
© Robert van Engelen, Florida State University
Alex Aiken and Sean Treichler, Stanford University
Compilers and Interpreters

- **Compilation**: Translation of a program written in a source language into a semantically equivalent program written in a target language.
Compilers and Interpreters (cont’d)

- **Interpretation**: Performing the operations implied by the source program
Compilers and Interpreters (cont’d)

- Compiler usually much faster than interpreter
- Interpreter can provide better error diagnostics, because of execution
Compilers and Interpreters (cont’d)

• **Hybrid models** combine compilation and interpretation
  
  – generate **bytecode** to be interpreted by a virtual machine (Java)
  
  – use **just-in-time (JIT)** compilation, also called dynamic translation
Compilers and Interpreters (cont’d)
Preprocessors, Compilers, Assemblers, and Linkers

Skeletal Source Program

Preprocessor

Source Program

Compiler

Target Assembly Program

Assembler

Relocatable Object Code

Linker/Loader

Absolute Machine Code

Libraries and Relocatable Object Files
The Analysis-Synthesis Model of Compilation

- There are two parts to compilation:
  - Analysis determines the operations implied by the source program which are recorded in a tree structure
  - Synthesis takes the tree structure and translates the operations therein into the target program
Other Tools that Use the Analysis-Synthesis Model

- Interpreters
- Editors (syntax highlighting)
- XML processors
- Text formatters (e.g. TeX and LaTeX)
- Silicon compilers (e.g. VHDL)
- Query interpreters/compilers (Databases)
- Natural Language Processing (Artificial Intelligence)
Compiler Phases

Character Stream
  ↓
Lexical Analyzer

Token Stream
  ↓
Syntax Analyzer

Syntax Tree
  ↓
Semantic Analyzer

Abstract Syntax Tree
  ↓
Intermediate Code Generator

Symbol Table

Intermediate Representation

(cont’d)
Compiler Phases (cont’d)

Intermediate Representation → Machine-Independent Code Optimizer

Intermediate Representation → Code Generator

Symbol Table

Target Machine Code → Machine-Dependent Code Optimizer

Target Machine Code
\[ position = initial + rate \times 60 \]

**Lexical Analyzer**

<table>
<thead>
<tr>
<th></th>
<th>position</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>initial</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>rate</td>
<td></td>
</tr>
</tbody>
</table>

**Symbol Table**
position = initial + rate * 60

Lexical Analyzer

⟨id, 1⟩ ⟨=⟩ ⟨id, 2⟩ ⟨+⟩ ⟨id, 3⟩ ⟨*⟩ ⟨60⟩
Syntax Analyzer

\[
S \quad = \quad T + F
\]

\[
T \quad = \quad \text{id, 2} + \text{id, 3} \times 60
\]
Semantic Analyzer

=  

<id, 1>  

+  

<id, 2>  

*  

<id, 3>  

int2float
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
Code Optimizer

\[
\begin{align*}
t1 &= \text{id3} \times 60.0 \\
\text{id1} &= \text{id2} + t1
\end{align*}
\]
LDF R2 id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
The Grouping of Phases

• Compiler divided in:
  – **Front end**: analysis (machine independent)
  – **Back end**: synthesis (machine dependent)

• All language specific knowledge must be encoded in the front-end

• All target specific knowledge must be encoded in the back-end
The Grouping of Phases (cont’d)

• Compiler passes:
  – A collection of phases is implemented in a single pass or in multi pass
  – Phases follow a *logical* division, passes follow an *operational* division
The Grouping of Phases (cont’d)
Compiler-Construction Tools

- Software development tools are available to implement one or more compiler phases
  - Scanner generators
  - Parser generators
  - Syntax-directed translation engines
  - Automatic code generators
  - Data-flow engines
Some History

- 1952 - IBM 701
  - 16K ops/second
  - 8KB memory
- All programming done in assembly
- Problem: **Software costs exceeded hardware costs!**
Speedcoding

• John Backus, 1953
• Allowed programs to use higher-level instructions
  – e.g. exp, log, sin in addition to add, sub, mult
• Ran as an interpreter
  – 10-20x slower than native execution
  – But better “time to solution” in many cases
Fortran I

• John Backus, 1954

• Idea
  – Translate high-level code to assembly
  – Many thought this impossible
  – Had already failed in other projects
Fortran I (cont’d)

- 1954-7: FORTRAN I project
- 1958: >50% of all software is in FORTRAN
- Development time halved
Fortran I (cont’d)

• The first compiler
• Huge impact on computer science
• Led to an enormous body of theoretical work
• Modern compilers preserve the outlines of FORTRAN I
Modern Compilers

• The overall structure of almost every compiler adheres to our outline

• The proportions have changed since Fortran
  – Early: lexing, parsing most complex, expensive
  – Today: optimization dominates all other phases, lexing and parsing are cheap
Impacts on Compilers

- Design of programming languages and compilers intimately related
- Compilers also used as a tool in evaluating computer architectures
Impacts on Compilers (cont’d)

• Compilers also show a beautiful example of how theory meets practice

• The study of compilers is a study of how we choose the right mathematical model (abstraction)
  – Finite-state machines
  – Context-free grammars
  – Optimization
Why So Many Languages?

• Application domains have distinctive and conflicting needs

• Examples:
  – data being manipulated, operations needed
  – productivity
  – performance
  – safety
  – extensibility
  – embeddability
Programming Language Economics

• Languages are adopted to fill a void
  – Enable a previously difficult/impossible application
  – Orthogonal to language design quality (almost)

• Programmer training is the dominant cost
  – **Start condition** for new languages: productivity > training cost
  – Languages with many users are replaced rarely
  – Easy to start in a new niche