Introduction

Chapter 1

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)

• “Hybrid model”
  – Combine compilation and interpretation
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)
Preprocessors, Compilers, Assemblers, and Linkers

Skeletal Source Program

Preprocessor

Source Program

Compiler

Target Assembly Program

Assembler

Relocatable Object Code

Linker/Loader

Libraries and Relocatable Object Files

Absolute Machine Code

Compiler Phases

Character Stream

Lexical Analyzer

Token Stream

Syntax Analyzer

Syntax Tree

Semantic Analyzer

Syntax Tree

Intermediate Code Generator

Intermediate Representation

(cont’d)
Compiler Phases (cont’d)

Symbol Table

Intermediate Representation

Machine-Independent Code Optimizer

Intermediate Representation

Code Generator

Target Machine Code

Machine-Dependent Code Optimizer

Target Machine Code

**position = initial + rate * 60**

Lexical Analyzer

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>position</td>
<td>…</td>
</tr>
<tr>
<td>2</td>
<td>initial</td>
<td>…</td>
</tr>
<tr>
<td>3</td>
<td>rate</td>
<td>…</td>
</tr>
</tbody>
</table>
position = initial + rate * 60

Lexical Analyzer

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

Syntax Analyzer

=  

⟨id, 1⟩  +  

⟨id, 2⟩  *  

⟨id, 3⟩  60
Semantic Analyzer

Semantic Analyzer

\[
(id_1, 1) = (id_2, 2) \times (id_3, 3) \quad \text{int2float}
\]

60

Intermediate Code Generator

Intermediate Code Generator

\[
t1 = \text{int2float}(60)
t2 = id_3 \times t1
t3 = id_2 + t2
id_1 = t3
\]
Code Optimizer

t1 = id3 * 60.0
id1 = id2 + t1

Code Generator

LDF R2 id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
The Grouping of Phases

• Compiler *front* and *back ends*:
  – Front end: *analysis* (*machine independent*)
  – Back end: *synthesis* (*machine dependent*)

• Compiler *passes*:
  – A collection of phases is implemented in a *single pass* or in *multi pass*

The Grouping of Phases

• 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

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

Intermediate Representations

• Compiler stages need to communicate with each other
  – Usually use compiler-specific intermediate representations (or intermediate languages)
• Ordered in descending level of abstraction
  – Source language
  – Token stream
  – Abstract syntax tree
  – Basic blocks, three-address code
  – Target code
More About IRs

- Lower levels expose features hidden by higher levels
  - registers, memory layout
  - control flow (loops, if-elses vs. arbitrary jumps)
- But lower levels obscure high-level meaning
- Different transformations possible in different intermediate representations

Issues

- Compiling is almost this simple, but there are many pitfalls
- Example: How are erroneous programs handled?
- Language design has big impact on compiler
  - Determines what is easy and hard to compile
  - Course theme: many trade-offs in language design
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

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
  - Languages with many users are replaced rarely
  - Popular languages become ossified
  - But easy to start in a new niche …