Programming Languages

Lecture 2: Translators & Virtual Machines

Benjamin J. Keller

Department of Computer Science, Virginia Tech
Lecture Outline

- Abstraction
- Translators
- Compilers
- Interpreters
- Semantics
Languages as Abstraction

- Programming language creates a virtual machine for programmer
- Dijkstra: Originally we were obligated to write programs so that a computer could execute them. Now we write the programs and the computer has the obligation to understand and execute them.
- Progress in programming languages characterized by increasing support of abstraction
Virtual Machines

- Machine language provides “raw” machine to execute programs
- Even machine language abstracts away from physical nature of computation
- Higher level language presents programmer with a more abstract model of a computer
- Language creates the illusion of a more sophisticated virtual machine
• Virtual machines built-up using translators and other virtual machines

• (Virtual) Machine language is instruction set supported by translator

• Virtual machine determined by implementation
Problems

- Possibilities
  - Different implementors have different conceptions of virtual machine
  - Different computers
  - Implementors may choose different ways to simulate virtual machine

- Combined can lead to different implementations (on same machine)

- How can we ensure that all implementations have same semantics?
Explicit Virtual Machines

- Pascal P-code and P-machine
- Modula-2 M-code and M-machine
- Java Bytecode and Java virtual machine
- Prolog and Warren Abstract Machine
Translators

- Some high-level languages have special purpose processors (LISP machine)
- Most high-level languages must be translated
- Two types of translators
  - interpreter
  - compiler
- Usually combination of the two
Pure Interpreter

- Simulate virtual machine

Repeat
  Get next statement
  Determine action(s)
  Call routines to perform action(s)
Until done

Source
Program
Data

Interpreter

Virtual Mach.

Results
• Executing program
  1. Translates all units of program into object code (e.g., machine language)
  2. Link into single relocatable machine code
  3. Load into Memory
## Compilation vs Interpretation

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Interpreter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each statement translated once</td>
<td>Translate only if executed</td>
</tr>
<tr>
<td>Must compile</td>
<td>Run immediately</td>
</tr>
<tr>
<td>Faster execution</td>
<td>Allows more supportive environment</td>
</tr>
<tr>
<td>Only object code in memory when executing</td>
<td>Interpreter in memory</td>
</tr>
<tr>
<td>Object file likely large</td>
<td>Source likely smaller</td>
</tr>
</tbody>
</table>
Mixed Translation

- Rare to have pure compiler or pure interpreter
- Typically compile into easier to interpret form
- Example: abstract syntax tree (compact representation of parse tree)
- Usually go farther into intermediate code (Java Bytecode) and then interpret
Compilation

- Two primary phases
  1. Analysis: tokenize, parse, generate simple intermediate code (type checking)
  2. Synthesis: optimization (instructions in context), code generation, linking and loading
- To build portable compiler, separate into front- and back-end
  - Front-end does analysis and some optimization of intermediate code
  - Back-end does code generation and full optimization
Analysis Phase

1. Lexical analysis — break program into lexemes (identifiers, operation symbols, keywords, punctuation, comments, etc.) Convert lexeme into token consisting of lexeme and token type.

2. Syntactical analysis — Use formal grammar to parse program and build parse tree (maybe implicitly). Add identifiers to symbol table.

3. Semantic analysis — Update symbol table (e.g., add type info). Insert implicit information (resolve overloaded operations), error detection, type checking. Traverse parse tree generating intermediate code.
Synthesis Phase

1. Optimization — Find adjacent store-reload instructions, evaluate common sub-expressions, move static code out of loops, allocate registers, etc.

2. Code generation — Generate assembly or machine code (sometimes C)

3. Linking and Loading — Resolve external references between separately compiled units. Make sure that all references are relative to beginning of program. Load program into memory.
Symbol Table

- Contains
  1. identifier names
  2. kind of identifier (variable, array, procedure, formal parameter)
  3. type
  4. visibility
- Used to check for errors and code generation.
- May be thrown away at end of compilation
- Kept for error reporting or dynamic name generation
Semantics

- Meaning of program
  - Formal semantics (operational)
  - How would an interpreter for the language work on a virtual machine?

- Work with virtual/abstract machine when discussing semantics of programming language constructs.
  - Represent \textit{data} and \textit{code} portions of memory
  - Instruction pointer, \textit{ip}, incremented after each instruction if not explicitly modified

- Run program by loading it into memory and initializing \textit{ip} to beginning of program.
Official Language Definitions

- Standardize syntax and semantics to promote portability
- All compilers should accept the same programs
- All legal programs should give the same answers (modulo differences in arithmetic, etc.)
- Designed for compiler writers and as programmer reference
Official Definitions (cont)

- Often better to standardize after gaining implementation experience (Ada standard before implemented)

- Common Lisp, Scheme, ML now standardized, Fortran 90, C++

- Standards include good formal definitions of syntax, but semantics still too hard

- Backus promised formal semantics in Algol 60 report — still waiting!