Compiling Programming Languages

Barrett R. Bryant

Software Composition & Modeling Laboratory

Department of Computer and Information Sciences
University of Alabama at Birmingham
Introduction

- All software running on all computers is written in some programming language.
- To be executed by a computer, a program must be translated into the machine language of that computer.
- A *compiler* is the software system that does this translation.
The von Neumann Architecture

Memory (stores both instructions and data)

Central processing unit

Arithmetic and logic unit

Control unit

Results of operations

Instructions and data

Input and output devices

Execution of Machine Code by Hardware Interpreter

• Fetch–execute–cycle

initialize the program counter

repeat forever
  fetch the instruction pointed by the counter
  increment the counter
  decode the instruction
  execute the instruction

end repeat
Evolution of Programming Languages

- Machine Language – 0's and 1's
- Assembly Language – mnemonic form of Machine Language
- First Generation Languages – higher-level data and control constructions corresponding to Machine Language data and control (e.g. FORTRAN)
- Second Generation Languages – higher-level data and control constructions, not always corresponding to, but still modeled after Machine Language data and control (e.g. ALGOL 60, COBOL)
- Third Generation Languages – introduction of more abstract forms of data, including user-defined data types (e.g. Pascal, C)
- Object-Based Languages – support for objects and abstract data types (e.g. Ada)
- Object-Oriented Languages – support for classes of objects organized as a class hierarchy (e.g. Smalltalk, C++, Java)
- Natural Languages – humans communicate directly with the machine (e.g. English)
Implementation Methods

• Compilation
  – Programs are translated into machine language, which is then executed by the hardware interpreter

• Pure Interpretation
  – Programs are interpreted by another program known as a software interpreter

• Hybrid Implementation Systems
  – A compromise between compilers and pure interpreters

Compilation

Implementation Methods

• Compilation
  – Programs are translated into machine language, which is then executed by the hardware interpreter

• Pure Interpretation
  – Programs are interpreted by another program known as a software interpreter

• Hybrid Implementation Systems
  – A compromise between compilers and pure interpreters

Pure Interpretation

Pure Interpretation

Implementation Methods

• Compilation
  – Programs are translated into machine language, which is then executed by the hardware interpreter

• Pure Interpretation
  – Programs are interpreted by another program known as a software interpreter

• Hybrid Implementation Systems
  – A compromise between compilers and pure interpreters

Hybrid Implementation

Layered View of Computer

The operating system and language implementation are layered over the machine interface of the underlying computer. Each language runs on its own virtual machine.

Compilation

• Translate high–level program (source language) into machine code (machine language)
• Slow translation, fast execution
• Compilation process has several phases:
  – lexical analysis: converts characters in the source program into lexical units
  – syntax analysis: transforms lexical units into *parse trees* which represent the syntactic structure of program
  – semantics analysis: generate intermediate code
  – code generation: machine code is generated

The Compilation Process

Front-End of a Compiler

Back–End of a Compiler

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3

code optimizer

temp1 := id3 * 60.0
id1 := id2 + temp1

code generator

MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

Additional Compilation Terminologies

- **Load module** (executable image): the user and system code together
- **Linking and loading**: the process of collecting system program and linking them to user program

Pure Interpretation

- No translation
- Easier implementation of programs (run-time errors can easily and immediately be displayed)
- Slower execution (10 to 100 times slower than compiled programs)
- Often requires more space
- Now rare for traditional high-level languages
- Significant comeback with some Web scripting languages (e.g., JavaScript, PHP)

Pure Interpretation Process

Source program

Interpreter

Input data

Results

Hybrid Implementation Systems

- A compromise between compilers and pure interpreters
- A high-level language program is translated to an intermediate language that allows easy interpretation
- Faster than pure interpretation
- Examples
  - Perl programs are partially compiled to detect errors before interpretation
  - Initial implementations of Java were hybrid; the intermediate form, *byte code*, provides portability to any machine that has a byte code interpreter and a run-time system (together, these are called *Java Virtual Machine*)

Hybrid Implementation Process

Just-in-Time Implementation Systems

- Initially translate programs to an intermediate language
- Then compile the intermediate language of the subprograms into machine code when they are called
- Machine code version is kept for subsequent calls
- JIT systems are widely used for Java programs
- .NET languages are implemented with a JIT system
  - All such languages are translated to a Common Intermediate Language (CIL) whose virtual machine is called the Common Language Run-Time (CLR)

Execution of a Java Applet

Preprocessors

- Preprocessor macros (instructions) are commonly used to specify that code from another file is to be included.
- A preprocessor processes a program immediately before the program is compiled to expand embedded preprocessor macros.
- A well-known example: C preprocessor expands #include, #define, and similar macros.

Compiler Construction Tools

- **Scanner generators** – produce lexical analyzers from regular expression descriptions of tokens
- **Parser generators** – produce syntax analyzers from grammars
- **Syntax-directed translation engines** – produce collections of routines for walking a parse tree and generating intermediate code
Scanner Generation

Specification of Tokens
(Regular Expressions)

Lexical Analyzer Generator (e.g., Jflex)

Lexical Analyzer
(Finite Automaton)
JFlex Example

Identifier = [:letter:] [:letter: | :digit:]*
Integer = [:digit:] [:digit:]*

[ 	
] { echo (); }
";"  { echo (); return new Token (Token.SEMICOLON); }
"."  { echo (); return new Token (Token.PERIOD); }
"<"  { echo (); return new Token (Token.RELOP, Token.LT); }
">
"="  { echo (); return new Token (Token.RELOP, Token.EQ); }
"+"  { echo (); return new Token (Token.ADDOP, Token.PLUS); }
"*"  { echo (); return new Token (Token.MULTOP, Token.TIMES); }
if { echo (); return new Token (Token.IF); }
while { echo (); return new Token (Token.WHILE); }
{Integer}  { echo ();
             return new Token (Token.INTEGER, yytext ()); }
{Identifier} { echo ();
              return new Token (Token.ID, yytext ()); }
Parser Generation

Specification of Syntax
(Context-Free Grammar)

Parser Generator
(e.g., CUP)

Syntax Analyzer
(Pushdown Automaton)
CUP Example

program ::= block PERIOD ;

block ::= constDecl varDecl procDecl statement ;

constDecl ::= CONST constAssignmentList SEMICOLON | ;

constAssignmentList ::= ID EQ INTEGER | constAssignmentList COMMA ID EQ INTEGER ;

varDecl ::= VAR identList SEMICOLON | ;

identList ::= ID | identList COMMA ID ;

procDecl ::= procDecl PROC ID SEMICOLON block SEMICOLON | ;

statement ::= ID ASSIGN expression | BEGIN statementList END | IF condition THEN statement | WHILE condition DO statement | ;
Syntax–Directed Translation Engines

Specification of Semantics
(Attributed Context–Free Grammar)

Attribute Grammar Evaluator (e.g., LISA)

Semantic Analyzer and Intermediate Code Generator
Attribute Grammar Example

\[ <\text{term}> ::= <\text{factor}> \]

\[ <\text{factor}> . env ← <\text{term}> . env \]
\[ <\text{term}> . tree ← <\text{factor}> . tree \]
\[ <\text{term}> . type ← <\text{factor}> . type \]
\[ | <\text{term}>[1] <\text{multiplying-operator}> <\text{factor}> \]
\[ <\text{term}>[1] . env ← <\text{term}> . env \]
\[ <\text{factor}> . env ← <\text{term}> . env \]
\[ <\text{term}> . tree ← \]
\[ \text{tree} (<\text{multiplying-operator}> . \text{lexeme}, \]
\[ <\text{term}>[1] . \text{tree}, <\text{factor}> . \text{tree}) \]
\[ <\text{term}> . type ← \]
\[ \text{compatible} (<\text{term}>[1] . \text{type}, <\text{factor}> . \text{type}) \]
\[ <\text{multiplying-operator}> ::= * | / \]
Applications of Compiler Technology

• Implementation of High-Level Programming Languages
• Optimizations for Computer Architectures (e.g. parallelism, memory hierarchies)
• Design of New Computer Architectures (e.g. RISC, embedded systems)
• Program Translations (e.g. binary translation, hardware synthesis, database query interpreters)
• Software Productivity Tools (e.g. type/bounds checking, memory management)
A Grand Challenge for Computing Research

• A *verifying compiler* uses automated mathematical and logical reasoning to check the correctness of the programs that it compiles.

• The criterion of correctness is specified by types, assertions, and other redundant annotations that are associated with the code of the program.

Further Information

Primary References:


Contact Information:

bryant@cis.uab.edu

http://www.cis.uab.edu/softcom