An Overview of a Compiler

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design
Outline of the Lecture

- About the course
- Why should we study compiler design?
- Compiler overview with block diagrams
About the Course

- A detailed look at the internals of a compiler
- Does not assume any background but is intensive
- Doing programming assignments and solving theoretical problems are both essential
- A compiler is an excellent example of theory translated into practice in a remarkable way
Why Should We Study Compiler Design?

- Compilers are everywhere!
- Many applications for compiler technology
  - Parsers for HTML in web browser
  - Interpreters for javascript/flash
  - Machine code generation for high level languages
  - Software testing
  - Program optimization
  - Malicious code detection
  - Design of new computer architectures
    - Compiler-in-the-loop hardware development
  - Hardware synthesis: VHDL to RTL translation
  - Compiled simulation
    - Used to simulate designs written in VHDL
    - No interpretation of design, hence faster
A compiler is possibly the most complex system software and writing it is a substantial exercise in software engineering.

The complexity arises from the fact that it is required to map a programmer’s requirements (in a HLL program) to architectural details.

It uses algorithms and techniques from a very large number of areas in computer science.

Translates intricate theory into practice - enables tool building.
About the Nature of Compiler Algorithms

- Draws results from mathematical logic, lattice theory, linear algebra, probability, etc.
  - type checking, static analysis, dependence analysis and loop parallelization, cache analysis, etc.
- Makes practical application of
  - Greedy algorithms - register allocation
  - Heuristic search - list scheduling
  - Graph algorithms - dead code elimination, register allocation
  - Dynamic programming - instruction selection
  - Optimization techniques - instruction scheduling
  - Finite automata - lexical analysis
  - Pushdown automata - parsing
  - Fixed point algorithms - data-flow analysis
  - Complex data structures - symbol tables, parse trees, data dependence graphs
  - Computer architecture - machine code generation
Other Uses of Scanning and Parsing Techniques

- Assembler implementation
- Online text searching (GREP, AWK) and word processing
- Website filtering
- Command language interpreters
- Scripting language interpretation (Unix shell, Perl, Python)
- XML parsing and document tree construction
- Database query interpreters
Other Uses of Program Analysis Techniques

- Converting a sequential loop to a parallel loop
- Program analysis to determine if programs are data-race free
- Profiling programs to determine busy regions
- Program slicing
- Data-flow analysis approach to software testing
  - Uncovering errors along all paths
  - Dereferencing null pointers
  - Buffer overflows and memory leaks
- Worst Case Execution Time (WCET) estimation and energy analysis
Language Processing System

source program

Preprocessor

modified source program

Compiler

A LANGUAGE PROCESSING SYSTEM

target assembly program

Assembler

relocatable machine code

Linker/Loader

library files

relocatable object files

target machine code
Compiler Overview

- Character stream
  - Lexical Analyzer
    - Token stream
      - Syntax Analyzer
        - Syntax tree
          - Semantic Analyzer
            - Annotated syntax tree
              - Intermediate Code Generator
                - Intermediate representation
      - Machine-Dependent Code Optimizer
        - Target-machine code
          - Code Generator
            - Optimized intermediate representation
              - Machine-Independent Code Optimizer
                - Target-machine code

Compilers and Interpreters

- Compilers generate machine code, whereas interpreters interpret intermediate code.
- Interpreters are easier to write and can provide better error messages (symbol table is still available).
- Interpreters are at least 5 times slower than machine code generated by compilers.
- Interpreters also require much more memory than machine code generated by compilers.
- Examples: Perl, Python, Unix Shell, Java, BASIC, LISP.
fahrenheit = centigrade * 1.8 + 32

Lexical Analyzer

<id,1> <assign> <id,2> <multop> <fconst, 1.8> <addop> <iconst,32>

Syntax Analyzer
Lexical Analysis

- LA can be generated automatically from regular expression specifications
  - LEX and Flex are two such tools
- LA is a deterministic finite state automaton
- Why is LA separate from parsing?
  - Simplification of design - software engineering reason
  - I/O issues are limited LA alone
  - LA based on finite automata are more efficient to implement than pushdown automata used for parsing (due to stack)
Translation Overview - Syntax Analysis

\[
\langle \text{id,1} \rangle \langle \text{assign} \rangle \langle \text{id,2} \rangle \langle \text{multop} \rangle \\
\langle \text{fconst, 1.8} \rangle \langle \text{addop} \rangle \langle \text{iconst,32} \rangle
\]

Syntax Analyzer

\[
\begin{array}{c}
\text{id} \\
+ \\
\ast \\
1.8 \\
\text{id}
\end{array}
\]

Semantic Analyzer
Syntax analyzers (parsers) can be generated automatically from several variants of context-free grammar specifications
- LL(1), and LALR(1) are the most popular ones
- ANTLR (for LL(1)), YACC and Bison (for LALR(1)) are such tools

Parsers are deterministic push-down automata

Parsers cannot handle context-sensitive features of programming languages; e.g.,
- Variables are declared before use
- Types match on both sides of assignments
- Parameter types and number match in declaration and use
Translation Overview - Semantic Analysis

Syntax tree

Semantic Analyzer

Int. Code Generator

Y.N. Srikant
Semantic Analysis

- Semantic consistency that cannot be handled at the parsing stage is handled here
- Type checking of various programming language constructs is one of the most important tasks
- Stores type information in the symbol table or the syntax tree
  - Types of variables, function parameters, array dimensions, etc.
  - Used not only for semantic validation but also for subsequent phases of compilation
- Static semantics of programming languages can be specified using attribute grammars
Translation Overview - Intermediate Code Generation


\[
\begin{align*}
  & = \\
  & = id1 + \\
  & = id2 * 1.8 \\
  & = \text{intofloat}(32) \\
  & = t1 + t2 \\
  & = t3 \\
  & = id1
\end{align*}
\]

Int. Code Generator

Code Optimizer
Intermediate Code Generation

While generating machine code directly from source code is possible, it entails two problems:
- With $m$ languages and $n$ target machines, we need to write $m \times n$ compilers.
- The code optimizer which is one of the largest and very-difficult-to-write components of any compiler cannot be reused.

By converting source code to an intermediate code, a machine-independent code optimizer may be written.

Intermediate code must be easy to produce and easy to translate to machine code:
- A sort of universal assembly language.
- Should not contain any machine-specific parameters (registers, addresses, etc.)
Different Types of Intermediate Code

- The type of intermediate code deployed is based on the application.
- Quadruples, triples, indirect triples, abstract syntax trees are the classical forms used for machine-independent optimizations and machine code generation.
- Static Single Assignment form (SSA) is a recent form and enables more effective optimizations.
  - Conditional constant propagation and global value numbering are more effective on SSA.
- Program Dependence Graph (PDG) is useful in automatic parallelization, instruction scheduling, and software pipelining.
\( t_1 = id2 \times 1.8 \)
\( t_2 = \text{intofloat}(32) \)
\( t_3 = t_1 + t_2 \)
\( id1 = t_3 \)

---

**Code Optimizer**

\( t_1 = id2 \times 1.8 \)
\( id1 = t_1 + 32.0 \)

---

**Code Generator**
Intermediate code generation process introduces many inefficiencies
- Extra copies of variables, using variables instead of constants, repeated evaluation of expressions, etc.

Code optimization removes such inefficiencies and improves code

Improvement may be time, space, or power consumption

It changes the structure of programs, sometimes of beyond recognition
- Inlines functions, unrolls loops, eliminates some programmer-defined variables, etc.

Code optimization consists of a bunch of heuristics and percentage of improvement depends on programs (may be zero also)
Examples of Machine-Independent Optimizations

- Common sub-expression elimination
- Copy propagation
- Loop invariant code motion
- Partial redundancy elimination
- Induction variable elimination and strength reduction
- Code optimization needs information about the program
  - which expressions are being recomputed in a function?
  - which definitions reach a point?
- All such information is gathered through data-flow analysis
t1 = id2 * 1.8
id1 = t1 + 32.0

Code Generator

LDF R2, id2
MULF R2, R2, 1.8
ADDF R2, R2, 32.0
STF id1, R2
Code Generation

- Converts intermediate code to machine code
- Each intermediate code instruction may result in many machine instructions or vice-versa
- Must handle all aspects of machine architecture
  - Registers, pipelining, cache, multiple function units, etc.
- Generating efficient code is an NP-complete problem
  - Tree pattern matching-based strategies are among the best
  - Needs tree intermediate code
- Storage allocation decisions are made here
  - Register allocation and assignment are the most important problems
Machine-Dependent Optimizations

- Peephole optimizations
  - Analyze sequence of instructions in a small window (*peephole*) and using preset patterns, replace them with a more efficient sequence
  - Redundant instruction elimination
    - e.g., replace the sequence [LD A,R1][ST R1,A] by [LD A,R1]
  - Eliminate “jump to jump” instructions
  - Use machine idioms (use INC instead of LD and ADD)
- Instruction scheduling (reordering) to eliminate pipeline interlocks and to increase parallelism
- Trace scheduling to increase the size of basic blocks and increase parallelism
- Software pipelining to increase parallelism in loops