Compiler Design and Optimization

Mathematical Notes

October 27, 2025

Contents

1 Introduction to Compilers

1.1 What is a Compiler?

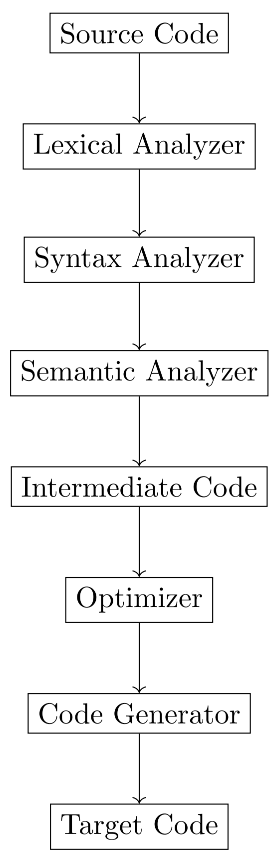
Definition 1.1 (Compiler)**.** *A compiler is a computer program that translates source code written in a high-level programming language into target code (usually machine code or bytecode) that can be executed by a computer.*

The compilation process typically involves several phases:

1. **Lexical Analysis** - Breaking source code into tokens
2. **Syntax Analysis** - Parsing tokens into abstract syntax trees
3. **Semantic Analysis** - Type checking and semantic validation
4. **Intermediate Code Generation** - Creating intermediate representation
5. **Code Optimization** - Improving the intermediate code
6. **Code Generation** - Producing target machine code

1.2 Compiler Architecture

Modern compilers typically follow a multi-pass architecture:



2 Lexical Analysis

2.1 Regular Expressions and Finite Automata

Definition 2.1 (Regular Expression). *A regular expression over alphabet Σ is defined recursively:*

- $\emptyset$ *(empty set)*
- ε *(empty string)*
- a *for* $a \in \Sigma$
- $r_1 + r_2$ *(union)*
- $r_1 \cdot r_2$ *(concatenation)*
- r^* *(Kleene star)*

Theorem 2.1 (Kleene's Theorem). *A language is regular if and only if it can be described by a regular expression.*

2.2 Finite State Automata

Definition 2.2 (Deterministic Finite Automaton (DFA)). *A DFA is a 5-tuple* $M = (Q, \Sigma, \delta, q_0, F)$ *where:*

- Q *is a finite set of states*
- Σ *is the input alphabet*
- $\delta : Q \times \Sigma \to Q$ *is the transition function*
- $q_0 \in Q$ *is the start state*
- $F \subseteq Q$ *is the set of accepting states*

Definition 2.3 (Non-deterministic Finite Automaton (NFA)). *An NFA is a 5-tuple* $M = (Q, \Sigma, \delta, q_0, F)$ *where:*

- Q *is a finite set of states*
- Σ *is the input alphabet*
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$ *is the transition function*
- $q_0 \in Q$ *is the start state*
- $F \subseteq Q$ *is the set of accepting states*

2.3 Lexical Analysis Implementation

The lexical analyzer (lexer) converts a stream of characters into a stream of tokens. Common token types include:

- Keywords (if, while, class, etc.)
- Identifiers (variable names)
- Literals (numbers, strings)
- Operators (+, -, *, /, etc.)
- Delimiters (parentheses, semicolons, etc.)

3 Syntax Analysis

3.1 Context-Free Grammars

Definition 3.1 (Context-Free Grammar)**.** *A context-free grammar is a 4-tuple* $G = (V, T, P, S)$ *where:*

- V *is a finite set of non-terminals*
- T *is a finite set of terminals*
- P *is a finite set of productions of the form* $A \to \alpha$
- $S \in V$ *is the start symbol*

Example 3.1 (Simple Arithmetic Grammar)**.**

$$E \to E + T \mid T \tag{1}$$
$$T \to T * F \mid F \tag{2}$$
$$F \to (E) \mid id \tag{3}$$

3.2 Parsing Algorithms

3.2.1 Top-Down Parsing

Definition 3.2 (LL(k) Grammar)**.** *A grammar is LL(k) if it can be parsed deterministically from left to right, producing a leftmost derivation, using k tokens of lookahead.*

Recursive Descent Parsing:

- Each non-terminal corresponds to a procedure
- Procedure bodies implement the grammar rules
- Requires LL(1) grammar for deterministic parsing

Predictive Parsing:

- Uses parsing table to make decisions
- Constructs FIRST and FOLLOW sets
- Eliminates left recursion and left factoring

3.2.2 Bottom-Up Parsing

Definition 3.3 (LR(k) Grammar)**.** *A grammar is LR(k) if it can be parsed deterministically from left to right, producing a rightmost derivation in reverse, using k tokens of lookahead.*

LR(1) Parsing:

- Uses canonical LR(1) items
- Constructs parsing table with ACTION and GOTO
- Handles reduce-reduce and shift-reduce conflicts

LALR(1) Parsing:

- Merges LR(1) states with same core
- Smaller parsing tables than LR(1)
- May introduce reduce-reduce conflicts

3.3 Abstract Syntax Trees

Definition 3.4 (Abstract Syntax Tree (AST))**.** *An AST is a tree representation of the syntactic structure of source code, where each node represents a construct occurring in the source code.*

The AST abstracts away from the concrete syntax and focuses on the essential structure of the program.

4 Semantic Analysis

4.1 Type Systems

Definition 4.1 (Type System)**.** *A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.*

4.1.1 Type Checking

Static Type Checking:

- Performed at compile time
- Catches type errors before execution
- Requires type annotations or type inference

Dynamic Type Checking:

- Performed at runtime
- More flexible but less efficient
- Runtime type errors possible

4.1.2 Type Inference

Definition 4.2 (Type Inference). *Type inference is the process of automatically determining the types of expressions in a program without explicit type annotations.*

Hindley-Milner Type System:

- Polymorphic type system
- Algorithm W for type inference
- Unification-based approach

4.2 Symbol Tables

Definition 4.3 (Symbol Table). *A symbol table is a data structure used by a compiler to keep track of semantic information about various source language constructs.*

Symbol tables typically store:

- Variable names and types
- Function signatures
- Class definitions
- Scope information

5 Intermediate Code Generation

5.1 Intermediate Representations

5.1.1 Three-Address Code

Definition 5.1 (Three-Address Code). *Three-address code is an intermediate representation where each instruction has at most one operator and three operands.*

Example 5.1 (Three-Address Code).

$$t_1 = a + b \tag{4}$$
$$t_2 = t_1 * c \tag{5}$$
$$d = t_2 \tag{6}$$

5.1.2 Static Single Assignment (SSA)

Definition 5.2 (Static Single Assignment). *In SSA form, each variable is assigned exactly once, and every use of a variable is dominated by its definition.*

SSA form enables:

- Efficient data-flow analysis
- Dead code elimination
- Constant propagation
- Register allocation

5.2 Control Flow Graphs

Definition 5.3 (Control Flow Graph)**.** *A control flow graph (CFG) is a directed graph where nodes represent basic blocks and edges represent control flow between blocks.*

Definition 5.4 (Basic Block)**.** *A basic block is a sequence of consecutive statements with a single entry point and a single exit point.*

6 Code Optimization

6.1 Optimization Levels

1. **Local Optimization** - Within basic blocks
2. **Global Optimization** - Across basic blocks
3. **Interprocedural Optimization** - Across procedure boundaries

6.2 Data-Flow Analysis

Definition 6.1 (Data-Flow Analysis)**.** *Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program.*

6.2.1 Reaching Definitions

Definition 6.2 (Reaching Definition)**.** *A definition d reaches a point p if there exists a path from d to p such that d is not killed along that path.*

The reaching definitions problem can be formulated as:

$$\text{REACH}[n] = \text{GEN}[n] \cup (\text{REACH}[m] - \text{KILL}[n]) \tag{7}$$

where m is a predecessor of n.

6.2.2 Live Variables

Definition 6.3 (Live Variable)**.** *A variable v is live at a point p if there exists a path from p to a use of v that does not redefine v.*

6.3 Optimization Techniques

6.3.1 Constant Folding and Propagation

Definition 6.4 (Constant Folding)**.** *Constant folding is the process of evaluating constant expressions at compile time.*

Definition 6.5 (Constant Propagation)**.** *Constant propagation is the process of replacing variables with their constant values when possible.*

6.3.2 Dead Code Elimination

Definition 6.6 (Dead Code). *Dead code is code that can never be executed or whose results are never used.*

Dead code elimination removes:

- Unreachable code
- Unused variables
- Unused functions

6.3.3 Common Subexpression Elimination

Definition 6.7 (Common Subexpression Elimination). *CSE identifies and eliminates redundant computations of the same expression.*

6.3.4 Loop Optimizations

Loop Invariant Code Motion:

- Moves computations outside loops
- Reduces redundant calculations

Loop Unrolling:

- Replicates loop body multiple times
- Reduces loop overhead
- Enables further optimizations

Induction Variable Elimination:

- Eliminates unnecessary induction variables
- Simplifies loop conditions

6.3.5 Function Inlining

Definition 6.8 (Function Inlining). *Function inlining replaces a function call with the body of the called function.*

Benefits:

- Eliminates call overhead
- Enables further optimizations
- May increase code size

7 Register Allocation

7.1 Graph Coloring

Definition 7.1 (Register Allocation). *Register allocation is the process of assigning program variables to processor registers.*

Theorem 7.1 (Graph Coloring Theorem). *A graph is k-colorable if and only if it does not contain a complete subgraph of size $k + 1$.*

7.2 Interference Graphs

Definition 7.2 (Interference Graph). *An interference graph is a graph where nodes represent variables and edges represent variables that cannot be assigned to the same register.*

Chaitin's Algorithm:

1. Build interference graph
2. Simplify graph by removing low-degree nodes
3. Spill nodes if graph is not colorable
4. Color the graph

7.3 Spilling

Definition 7.3 (Spilling). *Spilling is the process of storing variables in memory when there are not enough registers.*

Spill heuristics:

- Spill variables with high spill cost
- Spill variables with low usage frequency
- Consider loop nesting levels

8 Code Generation

8.1 Instruction Selection

Definition 8.1 (Instruction Selection). *Instruction selection is the process of choosing appropriate machine instructions to implement each intermediate code operation.*

Tree Pattern Matching:

- Represent instructions as tree patterns
- Use dynamic programming for optimal selection
- Handle complex addressing modes

8.2 Instruction Scheduling

Definition 8.2 (Instruction Scheduling). *Instruction scheduling is the process of reordering instructions to improve performance while maintaining correctness.*

Pipeline Scheduling:

- Minimize pipeline stalls
- Consider instruction latencies
- Handle data dependencies

8.3 Peephole Optimization

Definition 8.3 (Peephole Optimization). *Peephole optimization is a local optimization technique that examines a small window of instructions and replaces them with more efficient sequences.*

Common peephole optimizations:

- Redundant load elimination
- Strength reduction
- Branch optimization

9 Advanced Topics

9.1 Just-In-Time Compilation

Definition 9.1 (Just-In-Time Compilation). *JIT compilation is a method of executing computer code that involves compilation during program execution rather than before.*

JIT compilation benefits:

- Profile-guided optimization
- Adaptive optimization
- Runtime specialization

9.2 Parallel Compilation

Definition 9.2 (Parallel Compilation). *Parallel compilation distributes compilation tasks across multiple processors to reduce compilation time.*

Parallelization strategies:

- File-level parallelism
- Function-level parallelism
- Phase-level parallelism

9.3 Compiler Correctness

Definition 9.3 (Compiler Correctness). *A compiler is correct if it preserves the semantics of the source program in the target program.*

Verification techniques:

- Translation validation
- Formal verification
- Testing and validation

10 Modern Compiler Design

10.1 Multi-Pass Architecture

Modern compilers often use multiple passes for:

- Modularity and maintainability
- Different optimization levels
- Language-specific processing

10.2 Plugin Architecture

Definition 10.1 (Plugin Architecture). *A plugin architecture allows extending compiler functionality through dynamically loaded modules.*

Benefits:

- Language-specific optimizations
- Target-specific code generation
- Custom analysis passes

10.3 Compiler Infrastructure

Popular compiler infrastructures:

- **LLVM** - Low Level Virtual Machine
- **GCC** - GNU Compiler Collection
- **MLIR** - Multi-Level Intermediate Representation

11 Performance Analysis

11.1 Compilation Metrics

Key performance metrics:

- Compilation time
- Memory usage
- Generated code size
- Runtime performance

11.2 Optimization Effectiveness

Measuring optimization effectiveness:

- Benchmark suites
- Profiling tools
- Performance counters

12 Conclusion

Compiler design and optimization is a complex field that combines theoretical computer science with practical engineering. The key principles include:

- Modular design with clear separation of concerns
- Systematic application of optimization techniques
- Careful balance between compilation time and code quality
- Adaptation to modern hardware architectures

The field continues to evolve with new languages, architectures, and optimization techniques, making it an exciting area of computer science research and development.