Anatomy of a Compiler

Sam Caldwell

Champlain College

CSCI-430: Compilers

Dr. Ken Revett

28 October 2025

Anatomy of a Compiler

A modern compiler has a **front end** (analyze source), a **middle end** (optimize an intermediate representation), and a **back end** (produce target code). Cross-cutting facilities—**symbol tables**, **type systems**, **diagnostics**, and sometimes a **runtime**—support all stages (Aho, et. al, 2006; Cooper & Torczon, 2011).

Figure : Compiler Flowchart

A diagram of a diagram

AI-generated content may be incorrect.

# Major components and their roles

## Lexical analyzer (scanner)

The lexical analyzer converts a character stream to a token stream; handles whitespace/comments, recognizes keywords/identifiers/literals. This stage may handle simple error recovery/localization and track position for debugging (Aho et al., 2006).

For example, given the source line ‘let sum = 3 + count42;’, the lexical analyzer scans, partitioning the character stream into tokens. It recognizes ‘let’ as a keyword token, ‘sum’ as an identifier, ‘=’ as an assignment operator, ‘3’ as an integer literal, ‘+’ as an operator, ‘count42’ as an identifier, and ‘;’ as a delimiter. Whitespace is discarded. The scanner runs a deterministic finite automaton constructed from regular expressions; letters start an identifier state, digits start a number state, and punctuation drives operator or delimiter states. When a maximal munch rule would produce ambiguity, the DFA favors the longest match and, on ties, the rule with highest priority. As it emits tokens, the scanner annotates them with lexemes and source positions, enabling diagnostics. Errors such as an unterminated string or an invalid numeric escape are reported with recovery that skips to a safe boundary, a semicolon or newline. The resulting token stream is the sole input to the parser; comments and whitespace never reach syntax analysis. This partitioning enforces a clean separation of concerns and grounds subsequent phases in a regular-language view of the program used by later passes and optimizations (Aho, Lam, Sethi, & Ullman, 2006; Cooper & Torczon, 2011).

## Syntax analyzer (parser):

Tokens from the lexical analyzer are consumed by the syntax analyzer to build a parse tree, or Abstract Syntax Tree (AST), using a grammar (LL/LR variants). It detects and often recovers from syntactic errors (Aho et al., 2006). This AST is a tree-structured abstracted representation of a program’s source code in which each node denotes a language construct (e.g., expression, statement, declaration) and edges encode their hierarchical relationships. Unlike a concrete parse tree, an AST omits purely syntactic punctuation and reflects operator precedence and associativity directly in the shape of the tree. Compilers use the AST as the substrate for semantic analysis (name/type checking, symbol table annotation) and for lowering to intermediate representations. (Aho, Lam, Sethi, & Ullman, 2006; Cooper & Torczon, 2011).

Using the token stream from the above example (‘let sum = 3 + count42;’), the syntax analyzer creates the following Abstract Syntax Tree (AST):

Figure : Abstract Syntax Tree (AST)

|  |
| --- |
| SourceFile  `- LetDecl(name="sum")  `- Init  `- BinaryExpr(op="+")  |- IntLiteral(value=3)  `- Identifier(name="count42") |

This tree can be deconstructed to represent the original code or refined to more clearly represent the meaning of the program. At this point, the syntax (letter of the code) has been parsed into its structure, devoid of its actual meaning.

## Semantic analyzer:

The AST from the syntactical analysis is used by semantic analysis to determine and annotate the meaning of the program, resolving names, type checking, verifying scope/visibility rules and contextual constraints. Meaning is added to the AST through annotations and a symbol table is populated (Aho et al., 2006; Cooper & Torczon, 2011). The semantic analyzer reads the AST, applies the rules for the given language to determine meaning and updates the AST with annotations, also creating a symbols table, illustrated below:

Figure : Annotated AST and Symbol Table Example

|  |
| --- |
| SourceFile  `- LetDecl name="sum" : int [sym=S#1, mut, initialized]  `- Init : int [typecheck OK: int = int]  `- BinaryExpr(op="+") : int [rule: +(int, int)→int]  |- IntLiteral(value=3) : int [const, rvalue]  `- Identifier("count42") : int [sym=S#2, rvalue, resolved]  ;; Symbol table (relevant entries)  S#1: name=sum, type=int, storage=auto|global, scope=⟨current⟩, init=yes  S#2: name=count42, type=int, storage=unknown (predeclared), scope=⟨enclosing⟩  ;; Notes  - Name resolution: Identifier("count42") → S#2.  - Type checking: +(int, int) yields int; assigned to sum:int.  - No constant folding: right operand non-constant (identifier). |

The result of the above is the same source code with well-defined types and meaning which can be analyzed for correctness and used to generate the final output.

## IR generation and analysis:

IR generation lowers the typed AST to a machine-independent, analyzable program form. Expressions become simple operations; control flow becomes basic blocks and edges in a CFG. In SSA, each name has a single definition and φ-nodes merge values at joins, simplifying classic data-flow problems (Aho, Lam, Sethi, & Ullman, 2006; Cooper & Torczon, 2011). LLVM adopts a strongly typed, SSA-based IR organized as **modules → functions → basic blocks → instructions**, making data and control dependences explicit and portable across targets.

Figure : LLVM IR (SSA, typed) Example

|  |
| --- |
| @count42 = external global i32  @sum = external global i32  define void @f() {  entry:  %v1 = load i32, i32\* @count42  %v2 = add i32 3, %v1  store i32 %v2, i32\* @sum  ret void  } |

On this IR, analyses compute facts that justify semantics-preserving rewrites. Control-flow analyses (dominators, loop nests) and data-flow analyses (reaching definitions, liveness, available expressions) build def–use chains and identify optimization opportunities. Alias/points-to information restricts memory ambiguity; loop analyses expose induction variables and invariants. Guided by these results, transformations such as constant propagation, common subexpression elimination, dead code elimination, loop-invariant code motion, and strength reduction improve performance/size without changing meaning. LLVM’s pass pipeline embodies this mid-end: analyses (e.g., dominance, loop info, alias analysis) feed transforms (e.g., GVN, DCE, LICM), iterating to a fixpoint as needed. The optimized IR then flows to code generation for instruction selection, register allocation, and scheduling—separating machine-independent reasoning from target-specific lowering (Aho et al., 2006; Cooper & Torczon, 2011).

## IR transforms (optimizations, middle-end):

IR transforms **rewrite** the program to improve speed/size while preserving semantics. They are justified by analyses (dominators, data-flow, alias/points-to, loop info). Core transforms include **constant folding/propagation, common subexpression elimination (CSE/GVN), dead code elimination (DCE), copy/strength reduction, loop-invariant code motion (LICM), induction-variable simplification,** **inlining, scalar replacement of aggregates (SROA),** and **code motion/scheduling.** In SSA, def–use chains and φ-nodes make many of these simpler and more precise (Aho, Lam, Sethi, & Ullman, 2006; Cooper & Torczon, 2011). Below is an illustration of our earlier example, optimized:

Figure : LLVM-like example

|  |
| --- |
| entry:  %c = load i32, i32\* @c  br label %loop  loop:  %i = phi i32 [0, %entry], [%i2, %loop]  %p = getelementptr i32, i32\* %a, i32 %i  %v = load i32, i32\* %p  %m = mul i32 %v, %c  %s1 = add i32 %sum, %m  %i2 = add i32 %i, 1  %done = icmp eq i32 %i2, %n  br i1 %done, label %exit, label %loop |

Figure : After Optimization

|  |
| --- |
| preheader:  %c.pre = load i32, i32\* @c ; LICM: hoisted  br label %loop  loop:  %p = phi i32\* [%a, %preheader], [%p.next, %loop]  %v = load i32, i32\* %p  %m = mul i32 %v, %c.pre  %sum = add i32 %sum, %m  %p.next = getelementptr i32, i32\* %p, i32 1 ; strength reduction: pointer bump  ... (induction on pointer; dead temps removed) ... |

Hoisting removes repeated invariant loads; pointer bumping replaces %i\*4 addressing; SSA enables precise DCE/GVN across iterations. Net result: fewer memory ops, simpler address arithmetic, and tighter loops (Aho et al., 2006; Cooper & Torczon, 2011).

## Code generation (back end):

The optimized IR at this point is ready to be converted to a final output instruction set. This stage performs instruction selection (tree/graph covering), register allocation (graph coloring/linear scan), instruction scheduling, and emits target assembly/object code (Cooper & Torczon, 2011). In some languages, this phase will provide runtime services such as memory management, exception handling, stack/heap organization or a garbage collector (Cooper & Torczon, 2011).

# References

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools (2nd ed.). Addison-Wesley.

Cooper, K. D., & Torczon, L. (2011). Engineering a Compiler (2nd ed.). Morgan Kaufmann.