

Code Generation for Data Processing

Lecture 1: Introduction and Interpretation

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2025/26

Module “Code Generation for Data Processing”

Learning Goals

- ▶ Getting from an intermediate code representation to machine code
- ▶ Designing and implementing IRs and machine code generators
- ▶ Apply for: JIT compilation, query compilation, ISA emulation

Prerequisites

- ▶ Computer Architecture, Assembly ERA, GRA/ASP
- ▶ Databases, Relational Algebra GDB
- ▶ Beneficial: Compiler Construction, Modern DBs

Topic Overview

Introduction

- ▶ Introduction and Interpretation
- ▶ Compiler Front-end

Intermediate Representations

- ▶ IR Concepts and Design
- ▶ LLVM-IR
- ▶ Analyses and Optimizations

Compiler Back-end

- ▶ Instruction Selection
- ▶ Register Allocation
- ▶ Linker, Loader, Debuginfo

Applications

- ▶ JIT-compilation + Sandboxing
- ▶ Query Compilation
- ▶ Binary Translation

Lecture Organization

- ▶ Lecturer: Dr. Alexis Engelke `engelke@in.tum.de`
- ▶ Time slot: Thu 10-14, 02.11.018
- ▶ Material: <https://db.in.tum.de/teaching/ws2526/codegen/>

Exam

- ▶ Written exam, 90 minutes, **no retake**, 2026-02-27 14:00
- ▶ (Might change to oral on very low registration count)

Exercises

- ▶ Regular homework, often with programming exercise
- ▶ Submission via POST request (see assignments)
 - ▶ Grading with $\{*, +, \sim, -\}$, feedback on best effort
- ▶ Exercises integrated into lecture
- ▶ Hands-on programming or analysis of systems (needs laptop)
- ▶ Occasionally: present and discuss homework solutions

Grade Bonus

- ▶ Requirement: $N - 2$ “sufficiently working” homework submissions and one presentations of homework in class (depends on submission count)
- ▶ Bonus: grades in $[1.3; 4.0]$ improved by 0.3/0.4

Why study compilers?

- ▶ Critical component of every system, functionality and performance
 - ▶ Compiler mostly *alone* responsible for using hardware well
- ▶ Brings together many aspects of CS:
 - ▶ Theory, algorithms, systems, architecture, software engineering, (ML)
- ▶ New developments/requirements pose new challenges
 - ▶ New architectures, environments, language concepts, . . .
- ▶ High complexity!

Compiler Lectures @ TUM

Compiler Construction

IN2227, SS, THEO

Front-end, parsing, semantic analyses, types

Program Optimization

IN2053, WS, THEO

Analyses, transformations, abstract interpretation

Virtual Machines

IN2040, SS, THEO

Mapping programming paradigms to IR/bytecode

Programming Languages

CIT3230000, WS

Implementation of advanced language features

Code Generation

CIT3230001, WS

Back-end, machine code generation, JIT comp.

Why study code generation?

- ▶ Frameworks (LLVM, ...) exist and are comparably good, but often not good enough (performance, features)
 - ▶ Many systems with code gen. have their own back-end
 - ▶ E.g.: V8, WebKit FTL, .NET RyuJIT, GHC, Zig, QEMU, Umbra, ...
- ▶ Machine code is not the only target: bytecode
 - ▶ Often used for code execution
 - ▶ E.g.: V8, Java, .NET MSIL, BEAM (Erlang), Python, MonetDB, eBPF, ...
 - ▶ Allows for flexible design
 - ▶ But: efficient execution needs machine code generation

Proebsting's Law

“Compiler advances double computing power every *18* years.”

– Todd Proebsting, 1998¹

- ▶ Still optimistic; depends on number of abstractions

¹<http://proebsting.cs.arizona.edu/law.html>

Motivational Example: Brainfuck

- ▶ Turing-complete esoteric programming language, 8 operations
 - ▶ Input/output: . ,
 - ▶ Moving pointer over infinite array: < >
 - ▶ Increment/decrement: + -
 - ▶ Jump to matching bracket if (not) zero: []

++++++[->++++++<]>.

- ▶ Execution with pen/paper? ☹

Program Execution



Programs

- ▶ High flexibility (possibly)
- ▶ Many abstractions (typically)
- ▶ Several paradigms

Hardware/ISA

- ▶ Low-level interface
- ▶ Few operations, imperative
- ▶ “Not easy” to write

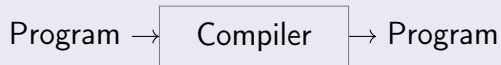
Motivational Example: Brainfuck – Interpretation

► Write an interpreter!

```
unsigned char state[10000];
unsigned ptr = 0, pc = 0;
while (prog[pc])
    switch (prog[pc++]) {
        case '.': putchar(state[ptr]); break;
        case ',': state[ptr] = getchar(); break;
        case '>': ptr++; break;
        case '<': ptr--; break;
        case '+': state[ptr]++; break;
        case '-': state[ptr]--; break;
        case '[': state[ptr] || (pc = matchParen(pc, prog)); break;
        case ']': state[ptr] && (pc = matchParen(pc, prog)); break;
    }
```

Program Execution

Compiler



- ▶ Translate program to other lang.
- ▶ Might optimize/improve program
- ▶ C, C++, Rust → machine code
- ▶ Python, Java → bytecode

Interpreter



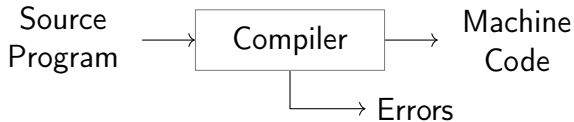
- ▶ Directly execute program
- ▶ Computes program result
- ▶ Shell scripts, Python bytecode, machine code (conceptually)

Multiple compilation steps can precede the “final interpretation”

Compilers

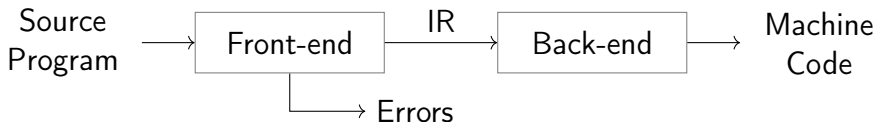
- ▶ Targets: machine code, bytecode, or other source language
- ▶ Typical goals: better language usability and performance
 - ▶ Make lang. usable at all, faster, use less resources, etc.
- ▶ Constraints: specs, resources (comp.-time, etc.), requirements (perf., etc.)
- ▶ Examples:
 - ▶ “Classic” compilers source → machine code
 - ▶ JIT compilation of JavaScript, WebAssembly, Java bytecode, ...
 - ▶ Database query compilation
 - ▶ ISA emulation/binary translation

Compiler Structure: Monolithic



- ▶ Inflexible architecture, hard to retarget

Compiler Structure: Two-phase architecture



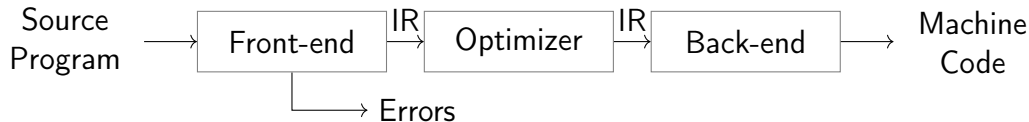
Front-end

- ▶ Parses source code
- ▶ Detect syntax/semantical errors
- ▶ Emit *intermediate representation* encode semantics/knowledge
- ▶ Typically: $\mathcal{O}(n)$ or $\mathcal{O}(n \log n)$

Back-end

- ▶ Translate IR to target architecture
- ▶ Can assume valid IR ($\rightsquigarrow$ no errors)
- ▶ Possibly one back-end per arch.
- ▶ Contains $\mathcal{NP}$ -complete problems

Compiler Structure: Three-phase architecture



- ▶ Optimizer: analyze/transform/rewrite program inside IR
-
- ▶ Conceptual architecture: real compilers typically much more complex
 - ▶ Several IRs in front-end and back-end, optimizations on different IRs
 - ▶ Multiple front-ends for different languages
 - ▶ Multiple back-ends for different architectures

Compiler Front-end

1. Tokenizer: recognize words, numbers, operators, etc. *Re*
 - ▶ Example: $a+b*c \rightarrow \text{ID}(a) \text{ PLUS ID}(b) \text{ TIMES ID}(c)$
2. Parser: build (abstract) syntax tree, check for syntax errors *CFG*
 - ▶ Syntax Tree: describe grammatical structure of complete program
Example: `expr("a", op("+"), expr("b", op("*"), expr("c")))`
 - ▶ Abstract Syntax Tree: only relevant information, more concise
Example: `plus("a", times("b", "c"))`
3. Semantic Analysis: check types, variable existence, etc.
4. IR Generator: produce IR for next stage
 - ▶ This might be the AST itself

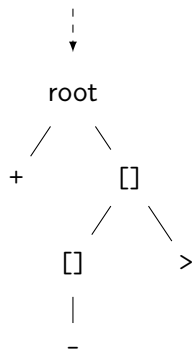
Compiler Back-end

1. Instruction Selection: map IR operations to target instructions
 - ▶ Use target features: special insts., addressing modes, ...
 - ▶ Still using virtual/unlimited registers
2. Instruction Scheduling: optimize order for target arch.
 - ▶ Start memory/high-latency earlier, etc.
 - ▶ Requires knowledge about micro-architecture
3. Register Allocation: map values to fixed register set/stack
 - ▶ Use available registers effectively, minimize stack usage

Motivational Example: Brainfuck – Front-end

- ▶ Need to skip comments
- ▶ Bracket searching is expensive/redundant
- ▶ Idea: “parse” program!
- ▶ Tokenizer: yield next operation, skipping comments
- ▶ Parser: find matching brackets, construct AST

+ [[-]>]



Motivational Example: Brainfuck – AST Interpretation

- ▶ AST can be interpreted recursively

```
struct node { char kind; unsigned cldCnt; struct node* cld; };
struct state { unsigned char* arr; size_t ptr; };
void donode(struct node* n, struct state* s) {
    switch (n->kind) {
        case '+': s->arr[s->ptr]++; break;
        // ...
        case '[': while (s->arr[s->ptr]) children(n, s); break;
        case 0: children(n, s); break; // root
    }
}
void children(struct node* n, struct state* s) {
    for (unsigned i = 0; i < n->cldCnt; i++) donode(n->cld + i, s);
}
```

Motivational Example: Brainfuck – Optimization

- ▶ Inefficient sequences of `+/-/</>` can be combined
 - ▶ Trivially done when generating IR
- ▶ Fold patterns into more high-level operations

Look at some Brainfuck programs. Which patterns are beneficial to fold?

Motivational Example: Brainfuck – Optimization

- ▶ Fold offset into operation
 - ▶ `right(2) add(1) = addoff(2, 1) right(2)`
 - ▶ Also possible with loops
- ▶ Analysis: does loop move pointer?
 - ▶ Loops that keep position intact allow more optimizations
 - ▶ Maybe distinguish “regular loops” from arbitrary loops?
- ▶ Get rid of all “effect-less” pointer movements
- ▶ Combine arithmetic operations, disambiguate addresses, etc.

Motivational Example: Brainfuck – Bytecode

- ▶ Tree is nice, but rather inefficient $\rightsquigarrow$ flat and compact bytecode
- ▶ Avoid pointer dereferences/indirections; keep code size small
- ▶ Maybe dispatch two instructions at once?
 - ▶ `switch (ops[pc] | ops[pc+1] << 8)`
- ▶ Superinstructions: combine common sequences to one instruction

Threaded Interpretation²

- ▶ Simple switch-case dispatch has lots of branch misses
- ▶ Threaded interpretation: at end of a handler, jump to next op

```
struct op { char op; char data; };
struct state { unsigned char* arr; size_t ptr; };
void threadedInterp(struct op* ops, struct state* s) {
    static const void* table[] = { &&CASE_ADD, &&CASE_RIGHT, };
#define DISPATCH do { goto *table[(++pc)->op]; } while (0)

    struct op* pc = ops;
    DISPATCH;

CASE_ADD: s->arr[s->ptr] += pc->data; DISPATCH;
CASE_RIGHT: s->arr += pc->data; DISPATCH;
}
```

²MA Ertl and D Gregg. "The structure and performance of efficient interpreters". In: *JILP* 5 (2003), pp. 1–25. .

Threaded Interpretation with Tail Calls

- Threaded interpretation can also be implemented with tails calls

```
struct op { char op; char data; };
struct state { unsigned char* arr; size_t ptr; };
void tcInterp(struct op* pc, struct state* s);
static void fn_add(struct op* pc, struct state* s) {
    s->arr[s->ptr] += pc->data; tcInterp(pc + 1, s); }
static void fn_right(struct op* pc, struct state* s) {
    s->arr += pc->data; tcInterp(pc + 1, s); }
void tcInterp(struct op* pc, struct state* s) {
    typedef void (* Fn)(struct op* pc, struct state* s);
    static const Fn fns[] = { fn_add, fn_right };
    fns[pc->op](pc, s);
}
```

Direct Threading⁵

- Use function pointer as operation to avoid indirection

```
struct op; struct state;
typedef void (*Fn)(struct op* pc, struct state* s);
struct op { Fn op; char data; };
struct state { unsigned char* arr; size_t ptr; };

void fn_add(struct op* pc, struct state* s) {
    s->arr[s->ptr] += pc->data; pc[1].op(pc + 1, s); }
void fn_right(struct op* pc, struct state* s) {
    s->arr += pc->data; pc[1].op(pc + 1, s); }
void dtInterp(struct op* ops, struct state* s) {
    ops[0].op(ops, s);
}
```

⁵JR Bell. "Threaded Code". In: *CACM* 16.6 (1973), pp. 370–372. .

Threaded Interpretation — Comparison

What are benefits/drawbacks of the three threading approaches⁶?

- ▶ Indirect threading with computed goto
 - ▶ Indirect threading with tail calls
 - ▶ Direct threading with tail calls
-
- ▶ Some differences on code size, readability, and maintainability
 - ▶ Performance: depends on hardware, context – always measure!

⁶<https://godbolt.org/z/4rcEv4sqj>

Fast Interpretation

- ▶ Key technique to “avoid” compilation to machine code
- ▶ Preprocess program into efficiently executable bytecode
 - ▶ Easily identifiable opcode, homogeneous structure
 - ▶ Can be linear (fast to execute), but trees also work
 - ▶ Match bytecode ops with needed operations $\rightsquigarrow$ fewer instructions
 - ▶ Larger operations preferable, but not too many (prediction)
- ▶ Perhaps optimize – if it's worth the benefit
 - ▶ Fold constants, combine instructions, ...
 - ▶ Consider superinstructions for common sequences
- ▶ For very cold code: avoid transformations at all

Interpretation vs. Compilation

Fundamental benefits of compilation:

- ▶ Elimination of interpreter dispatch
 - ▶ Can be significant for bytecodes with many small operations
- ▶ Use of CPU registers for values across bytecode instructions
 - ▶ Interpreter: most values must be stored in memory
 - ▶ Register allocation can bring large improvements
- ▶ No fundamental benefit: optimizations
 - ▶ Many optimizations can also be applied on bytecode

Fundamental benefits of interpretation: simple, portable

Compiler: Surrounding – Compile-time

- ▶ Typical environment for a C/C++ compiler:



- ▶ Calling Convention: interface with other objects/libraries
- ▶ Build systems, dependencies, debuggers, etc.
- ▶ Compilation target machine (hardware, VM, etc.)

Compiler: Surrounding – Run-time

- ▶ OS interface (I/O, ...)
- ▶ Memory management (allocation, GC, ...)
- ▶ Parallelization, threads, ...
- ▶ VM for execution of virtual assembly (JVM, ...)
- ▶ Run-time type checking
- ▶ Error handling: exception unwinding, assertions, ...
- ▶ Reflection, RTTI

Motivational Example: Brainfuck – Runtime Environment

- ▶ Needs I/O for . and ,
- ▶ Error handling: unmatched brackets
- ▶ Memory management: infinitely-sized array

How to efficiently emulate an infinitely sized array?

Compilation point: AoT vs. JIT

Ahead-of-Time (AoT)

- ▶ All code has to be compiled
- ▶ No dynamic optimizations
- ▶ Compilation-time secondary concern

Just-in-Time (JIT)

- ▶ Compilation-time is critical
- ▶ Code can be compiled on-demand
 - ▶ Incremental optimization, too
- ▶ Handle cold code fast
- ▶ Dynamic specializations possible
- ▶ Allows for `eval()`

Various hybrid combinations possible

Introduction and Interpretation – Summary

- ▶ Compilation vs. interpretation and combinations
- ▶ Compilers are key to usable/performant languages
- ▶ Target language typically machine code or bytecode
- ▶ Three-phase architecture widely used
- ▶ Interpretation techniques: bytecode, threaded interpretation, ...
- ▶ JIT compilation imposes different constraints

Introduction and Interpretation – Questions

- ▶ What is typically compiled and what is interpreted? Why?
 - ▶ PostScript, C, JavaScript, HTML, SQL
- ▶ What are typical types of output languages of compilers?
- ▶ How does a compiler IR differ from the source input?
- ▶ What is the impact of the language paradigm on optimizations?
- ▶ What are important factors for an efficient interpreter?
- ▶ What are inherent benefits of compilation over interpretation?
- ▶ What are key differences between AoT and JIT compilation?