CS 335: An Overview of Compilation

Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2023-24-II

Executing Programs

Programming languages are an abstraction for describing computations

Control flow constructs and data abstraction

Advantages of high-level programming language abstractions

Fast prototyping, improved productivity, readability, maintainability, and debugging

The abstraction needs to be transferred to machine-executable form to be executed

A Bit of History

In the early 1950s, most programming was with assembly language

- Programmers were reluctant to use high-level programming languages for fear of lack of performance
  - Led to low programmer productivity and high cost of software development

In 1954, John Backus proposed a program that translated high-level expressions into native machine code for IBM 704 mainframe

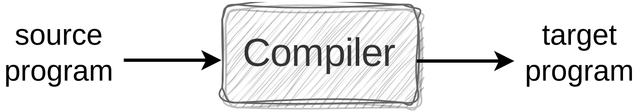
- Fortran (Formula Translator) I project (1954-1957): The first optimizing compiler was released
- The Fortran compiler has had a huge impact on the field of Programming Languages and Computer Science
  - Many advances in compilers were motivated by the need to generate efficient Fortran code
  - Modern compilers preserve the basic structure of the Fortran I compiler!

What is a Compiler?

Definition

A compiler is a **system software** that translates a program in a **source language** to an **equivalent** program in a **target language**

- System software (e.g., OS and compilers) helps application software (e.g., browser) to run
- Typical “source” languages might be C, C++, or Java
- The “target” language is usually the instruction set of some processor



Typesetting LaTeX source to generate PDF is an example of compilation

Important Goals of a Compiler

Generate correct code

- A compiler must preserve the meaning of the program being compiled
- Proving a compiler correct is a challenging problem and an active area of research

Must improve the code according to some metric

Performance or code size or energy consumption

Provide feedback to the user

Point out errors and potential mistakes in the program

Other concerns

- Compilation time and space required must be reasonable
- The engineering effort in building a compiler should be manageable

Automated Parallelization with Compiler Support

```c
// Disable optimizations
void serial(const float *A, const float *B, float *C) {
 for (int i = 0; i < N; i++) {
  C[i] = A[i] + B[i];
  C[i] = C[i] + C[i];
 }
}

void omp_parallel(const float * A, const float * B, float * C) {
 // Enable auto-parallelization with threads with OpenMP
 #pragma omp parallel for num_threads(omp_get_num_procs())
 for (int i = 0; i < N; i++) {
  C[i] = A[i] + B[i];
  C[i] = C[i] + C[i];
 }
}
```

Input file

Automated Parallelization with Compiler Support

```
// Disable optimizations
void serial(const float *A, const float *B, float *C) {
 for (int i = 0; i < N; i++) {
  C[i] = A[i] + B[i];
  C[i] = C[i] + C[i];
 }
```

```
❯ g++ -O0 -fopenmp omp-parallelization.cpp
❯ ./a.out
Reference Version: Vector Size = 268435456, Approximately 0.582 GFLOPS; Time = 0.923 sec
OpenMP Version: Vector Size = 268435456, Approximately 2.228 GFLOPS; Time = 0.241 sec
```

```
 for (int i = 0; i < N; i++) {
  C[i] = A[i] + B[i];
  C[i] = C[i] + C[i];
 }
}
```

Input file

Loop Transformations to Enable Parallelization

Thread Parallelism

```
// N and M are very large values

// Parallelize loop j with threads
for (int j = 1; j < N; j++) {
 for (int i = 1; i < M; i++) {
  A[i][j] = A[i-1][j] + B;
 }
}
```

Data Parallelism

```
// N and M are very large values

for (int i = 1; i < M; i++) {
 // Parallelize loop j with SIMD
 // instructions
 for (int j = 1; j < N; j++) {
  A[i][j] = A[i-1][j] + B;
 }
}
```

Source-to-Source Compiler

Produces a target program in **another programming language** rather than the assembly language of some processor

- Also known as transcompiler or transpiler
- TypeScript and CoffeeScript transpile to JavaScript, and many research compilers generate C programs
- The output programs require further translation before they can be executed

Compiler

- A **traditional** compiler translates a higher-level programming language to a lower-level language

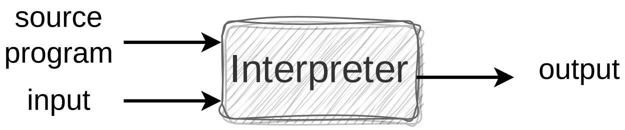
Transpiler

- Converts between programming languages at **approximately** the **same** level of abstraction

Interpreter

Definition

An **interpreter** takes as input an executable specification and produces as output the result of executing the specification



Scripting languages are often interpreted (e.g., Bash)

Compiler vs Interpreter

Compiler

- Translates the **whole** program at once
- Memory requirement during compilation is more
- Error reports are congregated
- On an error, compilers try to fix the error and proceed past
- Examples: C, C++, and Java

Interpreter

- Executes the program **one line** at a time
  - Compilation and execution happen at the same time
- Memory requirement is less because there is less state to maintain
- Error reports are per line, easier to report precise locations
- Stops translation on an error
- Examples: Bash and Python

More about Interpreters and Compilers

- Whether a language is interpreted or compiled is an **implementation-level** detail
  - If all implementations are interpreters, we say the language is interpreted
- Python is compiled to bytecode, and the bytecode is interpreted (CPython is the reference implementation)
  - Interpreting bytecode is faster than interpreting a higher-level representation
  - PyPy both interprets and just-in-time (JIT) compiles the bytecode to optimized machine code at run time

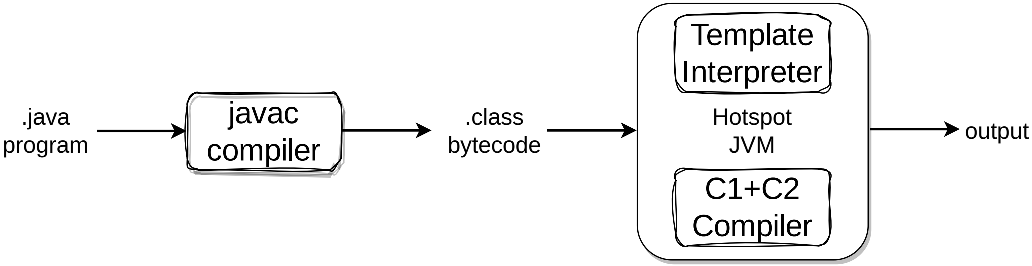
Is Python interpreted, or compiled, or both?

Hybrid Translation Schemes

- Translation process for a few languages includes both compilation and interpretation (e.g., Lisp)
- Java is compiled from source code into bytecode (`.class` files)
- Java virtual machines (JVMs) start execution by interpreting the bytecode
- JVMs include a just-in-time (JIT) compiler that compiles frequently-used bytecode sequences into native code
  - JIT compilation happens at run time and is driven by profiling
  - Important to keep the JIT compilation time low

How the JIT compiler boosts Java performance in OpenJDK

Compilation Flow in Java with Hotspot JVM

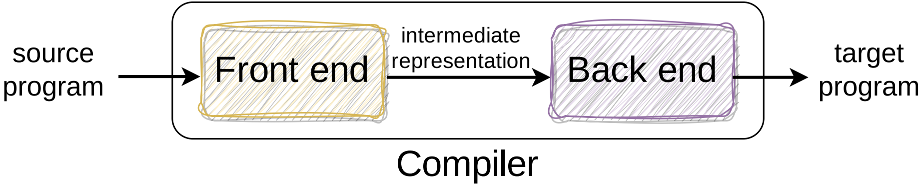


Demystifying the JVM: JVM Variants, Cppinterpreter and TemplateInterpreter

Structure of a Compiler

Compiler Structure

A compiler interfaces with both the source language and the target architecture



- Front end consists of two or three passes that handle the details of the input source-language program
- The back end passes lower the intermediate representation closer to the target machine's instruction set

Intermediate Representation

An intermediate representation (IR) is a data structure to encode information about the input program

- E.g., graphs, three address code, and LLVM IR
- Different IRs may be used during different phases of compilation

LLVM IR

```
clang -O0 -S -emit-llvm <file>.c
```

```c
int f(int a, int b) {
 return a + 2*b;
}
int main() {
 return f(10, 20);
}
```

$\xrightarrow[\text{LLVM IR}]{\text{Truncated}}$

```llvm
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @f(i32 noundef %0, i32 noundef %1) #0 {
 % 3 = alloca i32, align 4
 % 4 = alloca i32, align 4
 store i32 %0, i32* %3, align 4
 store i32 %1, i32* %4, align 4
 % 5 = load i32, i32* %3, align 4
 % 6 = load i32, i32* %4, align 4
 % 7 = mul nsw i32 2, %6
 % 8 = add nsw i32 %5, %7
 ret i32 %8
}
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
 %1 = alloca i32, align 4
 store i32 0, i32* %1, align 4
 %2 = call i32 @f(i32 noundef 10, i32 noundef 20)
 ret i32 %2
}
```

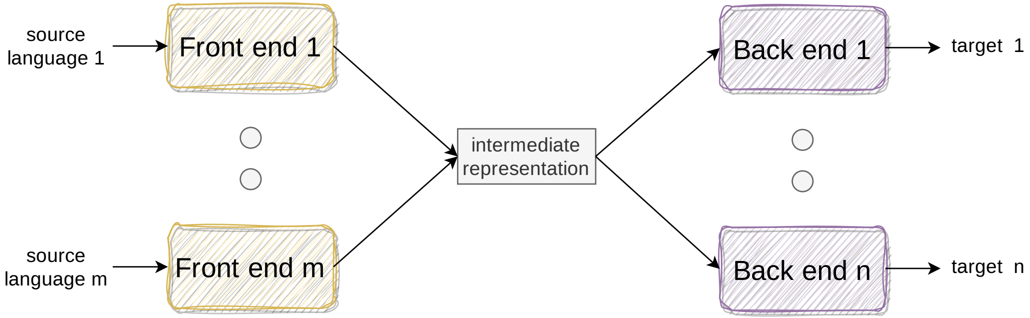
A Journey to understand LLVM-IR!

Learning LLVM

Advantages of Two-Phased Compiler Structure

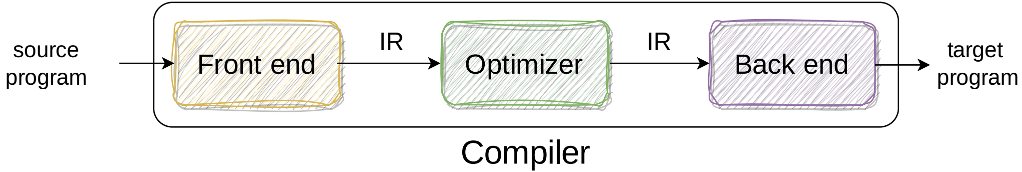
Simplifies the process of writing or retargeting a compiler

Retargeting is the task of adapting the compiler to generate code for a new processor



Three-Phased View of a Compiler

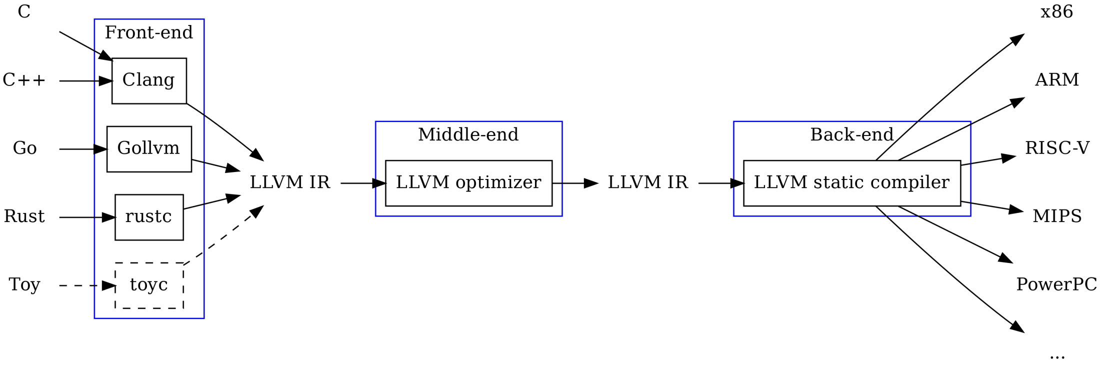
IR makes it possible to add more phases to compilation



Optimizer is an IR→IR transformer that tries to improve the IR program in some way

Optimization phase contains many passes to perform different optimizations

Visualizing the LLVM Compiler System



LLVM IR and Go

Implementing a Compiler

- A compiler is one of the most intricate software systems
  - General-purpose compilers often involve more than a hundred thousand LoC
- Very practical demonstration of the integration of theory and engineering

Idea	**Implementation**
Finite and push-down automata	Lexical and syntax analysis
Greedy algorithms	Register allocation
Fixed-point algorithms	Dataflow analysis
…	…

- Other practical issues such as ensuring concurrency, managing synchronization, and optimizing for the memory hierarchy and target processor complicate the implementation

Implementation Choices

Monolithic Design

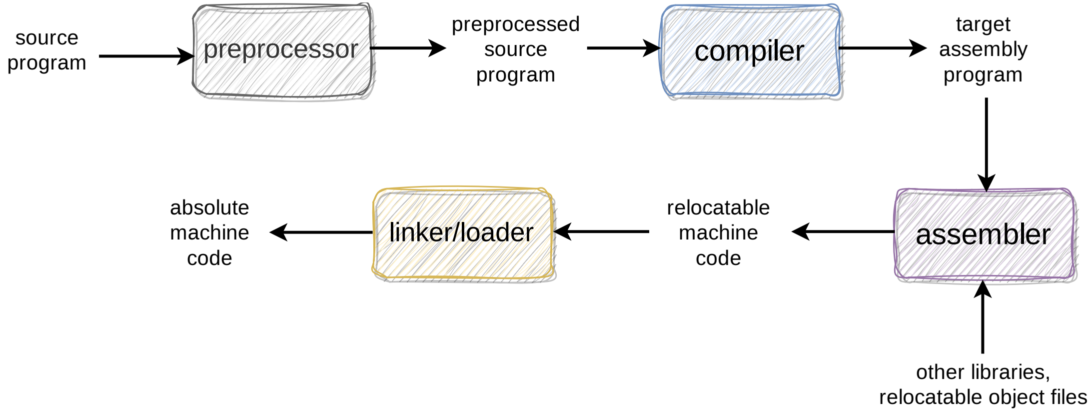
- Potentially more efficient but is less flexible

Multipass Design

- Less complex and easier to debug compiler bugs
- Can suffer from higher compilation times

Phases in a Compiler

Compiler Toolchain



Different Compilation Stages with gcc

Preprocess, do not compile: `gcc -E <file>.c -o <file>.i`

Invoke the C preprocessor directly: `cpp <file>.c -o <file>.i`

Compile, do not assemble: `gcc -S <file>.i -o <file>.s`

Invoke the compiler directly: `cc -S <file>.i -o <file>.s`

Compile to relocatable object file, do not link: `gcc -c <file>.s -o <file>.o`

Invoke the assembler directly: `as <file.s> -o <file.o>`

Link object file(s) to create executable: `gcc <file1>.o <file2>.o -o <file>`

Invoke the linker directly: `ld <file1>.o <file2.o> -o <file>`

Save intermediate files during compilation: `gcc -save-temps <file>.c -o <file>`

See commands invoked: `gcc -v <file>.c -o <file>.o`

Translation in a Compiler

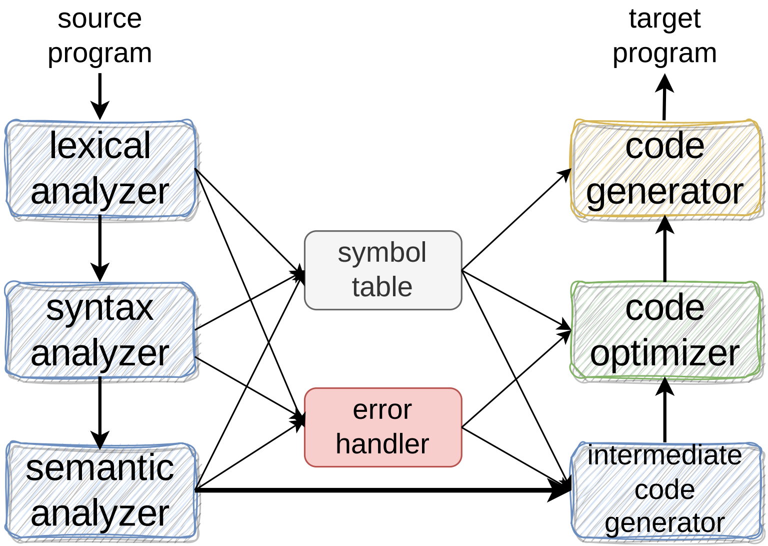
Direct translation from a high-level language to machine code is difficult

- Mismatch in the abstraction level between source code and machine code
  - Abstract data types and variables vs memory locations and registers
  - Control flow constructs vs jump and returns
- Some languages are farther from machine code than others (e.g., object-oriented languages)

Translate in small steps, where each step handles a reasonably simple, logical, and well-defined task

- Design a series of IRs to encode information across steps
  - IR should be amenable to program manipulation of various kinds (e.g., type checking, optimization, and code generation)
- IR becomes more machine-specific and less language-specific as translation proceeds

Different Phases in a Compiler



Front End

- The first step in translation is to compare the input program structure with the language definition
- Requires a formal definition of the language, in the form of **regular expressions** and **context-free grammar**
- Two separate passes in the front end, often called the **scanner** and the **parser**, determine whether or not the input code is a valid program defined by the grammar

Lexical Analysis

Reads characters in the source program and groups them into a stream of **tokens** (or words)

- Tokens represent a syntactic category (e.g., keywords) and can be augmented with the lexical value

Example: `position = initial + rate * 60`

- Tokens are `ID`, “=”, `ID`, “+”, `ID`, “*”, and `CONSTANT`
- Character sequence forming a token is called a **lexeme** (e.g., `position` and `initial`)

Challenge is to identify word separators

- The language must define rules for breaking a sentence into a sequence of tokens
  - Normally, white spaces and punctuations are token separators in languages
  - In programming languages, a character from a different class may also be treated as a token separator

Programming Language vs Natural Language

Challenges with natural languages

- Interpretation of words or phrases evolves over time
  - "awful" meant worthy of awe and "bachelor" meant an young knight
- Allows ambiguous interpretations
  - "I saw someone on the hill with a telescope." or "I went to the bank."
  - "Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo." is grammatically correct

Programming languages have well-defined structures and interpretations and disallow ambiguity

- System software and application programs require **structured** input
  - Command line interface in Operating Systems, query language processing in Databases, and typesetting systems like LaTeX

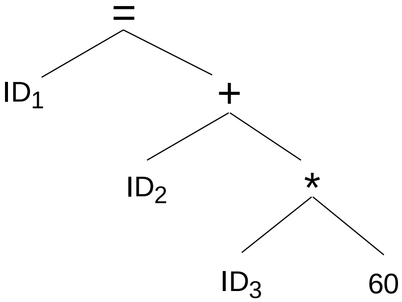
Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo

Syntax Analysis

Once tokens are formed, the next logical step is to understand the **structure of the sentence** via syntax analysis (or parsing)

- Syntax analysis imposes a hierarchical structure on the token stream

```
position = initial + rate * 60
```



Semantic Analysis

Once a sentence is constructed, semantic analysis interprets the **meaning** of the sentence

- Very challenging task for a compiler

```
X saw someone on the hill with a telescope.
```

```
JJ said JJ left JJ's assignment at home.
```

- Programming languages define very strict rules to avoid ambiguities (e.g., scope of variable `JJ`)
- Compilers perform other checks like type checking and matching formal and actual arguments of functions

```
position = initial + "rate" * 60
```

Intermediate Representation

- Once all checks pass, the front end generates an IR form of the code
- IR is a program for an abstract machine

$$\texttt{id}_1 = \texttt{id}_2 + \texttt{id}_3 * 60 \quad \Longrightarrow \quad \begin{array}{l} \texttt{t}_1 = \texttt{inttofloat(60)} \\ \texttt{t}_2 = \texttt{id}_3 * \texttt{t}_1 \\ \texttt{t}_3 = \texttt{t}_2 + \texttt{id}_2 \\ \texttt{id}_1 = \texttt{t}_3 \end{array}$$

Code Optimization

- Attempts to **improve** the IR code according to some metric
  - Reduce the execution time, code size, or resource usage
- "Optimizing" compilers spend a significant amount of compilation time in this phase
- Most optimizations consist of an **analysis** and a **transformation**
  - Analysis determines where the compiler can safely and profitably apply the technique
    - Data flow analysis tries to statically trace the flow of values at run time
    - Dependence analysis tries to estimate the possible values of array subscript expressions
- Example optimizations: Common sub-expression elimination, dead code elimination, loop invariant code motion, and constant folding

$$
\begin{array}{l}
t_1 = \mathtt{inttofloat(60)} \\
t_2 = id_3 * t_1 \\
t_3 = t_2 + id_2 \\
id_1 = t_3
\end{array}
\Longrightarrow
\begin{array}{l}
t_1 = id_3 * 60.0 \\
id_1 = t_1 + id_2
\end{array}
$$

Challenges with Code Optimization

- The same strategy may not work for all applications
  - Choice and order of optimizations
  - Parameters that control decisions and transformations
- Compiler may need to adapt its strategies to fit specific programs
- Active research on “autotuning” and “adaptive” runtimes
  - Compiler writer cannot predict a single answer for all possible programs
  - Use learning, models, or search to find good strategies

Code Generation

- Back end traverses the IR and **emits code** for the target machine
- The first stage is **instruction selection**
  - Translates IR operations into target machine instructions
  - Can take advantage of the feature set of the target machine
  - Assumes an infinite number of registers via virtual registers
- **Register allocation** decides which values should occupy the limited set of architectural registers
- **Instruction scheduling** reorders instructions to maximize utilization of hardware resources and minimize cycles

$$t_1 = id_3 * 60.0$$
$$id_1 = t_1 + id_2$$

$\Longrightarrow$

```
MOVSS id3, %XMM2 # load 32 bits
MULSS $60, %XMM2 # floating point
MOVSS id2, %XMM1
ADDSS %XMM2, %XMM1
MOVSS %XMM1, id1
```

Importance of Instruction Scheduling

Assume that `MOV` (i.e., memory access) takes 3 cycles, `MUL` takes 2 cycles, and `ADD` takes 1 cycle.

Naïve

```
MOVL off₁(addr₁), %R1
ADDL %R1, %R1
MOVL off₂(addr₂), %R2
MULL %R2, %R1
MOVL off₃(addr₃), %R3
MULL %R3, %R1
MOVL %R1, off₁(addr₁)
```

Improved

```
MOVL off₁(addr₁), %R1
MOVL off₂(addr₂), %R2
MOVL off₃(addr₃), %R3
ADDL %R1, %R1
MULL %R2, %R1
MULL %R3, %R1
MOVL %R1, off₁(addr₁)
```

References

- A. Aho et al. Compilers: Principles, Techniques, and Tools. Chapter 1, 2nd edition, Pearson Education.
- K. Cooper and L. Torczon. Engineering a Compiler. Chapter 1, 2nd edition, Morgan Kaufmann.