Compiler Construction

Chapter 1: Overview of Compilation

Dittaya Wanvarie

Department of Mathematics and Computer Science
Chulalongkorn University

Second semester, 2024

Outline

1. Course Outline
2. Introduction
3. The science of building a compiler
4. The structure of a compiler
   - The front end
   - The back end

Course outline

- Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022.
- `https://www.sciencedirect.com/book/9780128154120/engineering-a-compiler`

Grading policy

1. Midterm exam 30%
2. Final exam 30%
3. Assignments 40%

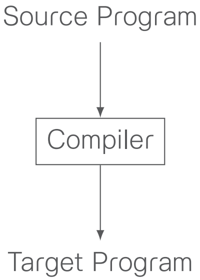
Outline

1. Course Outline
2. Introduction
3. The science of building a compiler
4. The structure of a compiler
   - The front end
   - The back end

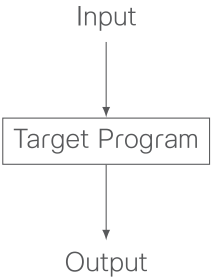
What does a compiler do?

Translate a program from one language - the source language - to another language - the target language

A compiler



Running a target program

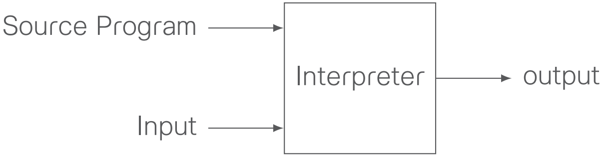


Compiler

- Translate text (source code) in one language into another language (binaries).
- Understand both the form, syntax, content, or meaning of the input language.
- Map the content from the source language to the target language

An interpreter

Interpreter is another language processor that *produces result*



Virtual machine

- A virtual machine is a simulator for some processor.
- An interpreter for that machine's instruction set

Instruction set

- The set of operations supported by a processor
- The overall design of an instruction set is often called an instruction set architecture or ISA
- Executable codes are codes using the instruction set

Overview of a compiler

Components

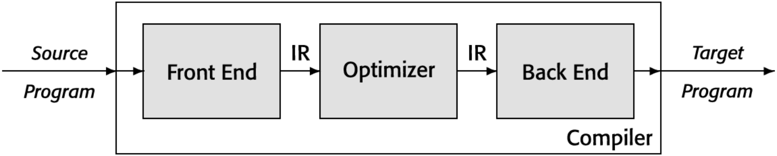


Figure: Components of a Compiler[1]

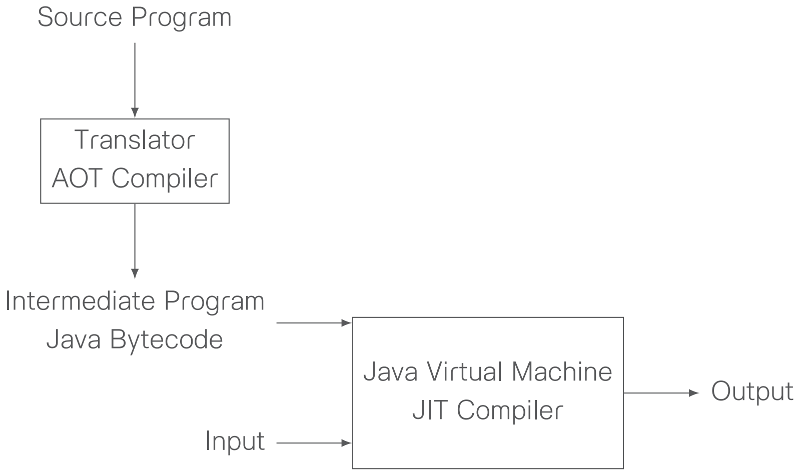
Compilation categories

- Ahead-of-time compiler: traditional
- Just-in-time compiler: adding cost to run-time

[1] Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022. Page 2.

Example: Java language processor

A hybrid compiler



A virtual machine is a simulator for some processor.

Other related programs

A language-processing system

Source Program

↓

Preprocessor — e.g. macro expansion

↓

Modified Source Program

↓

Compiler

↓

Target Assembly Program

↓

Assembler

↓

Relocatable machine code — e.g. binaries starting at address 0

↓

Linker/Loader ← Library files / Relocatable object files

↓

Target Machine Code — e.g. binaries starting at a real address in the memory

Outline

Why Study Compiler Construction?

A big software engineering projects that utilizes several basic knowledge in computer science

- Algorithms
  - Greedy algorithms (register allocation)
  - Heuristic search (list scheduling)
  - Graph algorithms (dead-code elimination)
  - Dynamic programming (instruction selection)
- Automata
  - Finite automata (scanner)
  - Push-down automata (parser)
- Dynamic allocation
- Synchronization
- Naming
- Locality and memory management
- Scheduling

Language features

New features in a programming language should be supported by the compiler

- Automatic register assignment
- User-defined data structure
- Control flow
- Data abstraction and inheritance of properties
- Type safety
- Garbage collection

Computer architectures

Optimizations of computer architectures

- Parallelism
- Memory hierarchies

Design of new computer architectures

- RISC
- Specialized architectures
  - VLIW, SIMD, etc.

Program translations

- Binary translation
- Hardware synthesis
- Database query interpreters: QL
- Compiled simulation
- Software productivity tools: data flow analysis
- Type checking
- Bounds checking: buffer overflow prevention
- Memory-management tools: Valgrind

Using Mathematics

Solving real-world problems **mathematically**

- Formulate the problem using mathematical abstraction
  - Finite-state machines, context-free grammars
- Solve the problem using mathematical techniques

Code optimization

- In general, a compiler cannot guarantee that one code is faster than other codes
- However, we can prove **mathematically** that an optimization is correct in some cases for all possible inputs

Why study compiler construction?

Do you know the result of a compiler translating our source code into machine instructions?

- We want to compare two strings, `s1` and `s2`
  1. `s1 == s2`
  2. `hash(s1) == hash(s2)`
- We have a multi-way selection with hundreds of cases
  1. Implementing a switch-case
  2. Implementing a hash for index in an GOTO array

Fundamental principles

1. The compiler must preserve the meaning of the program being compiled
2. The compiler must improve the input program in some discernible way

Code Optimization

Objectives of the optimization

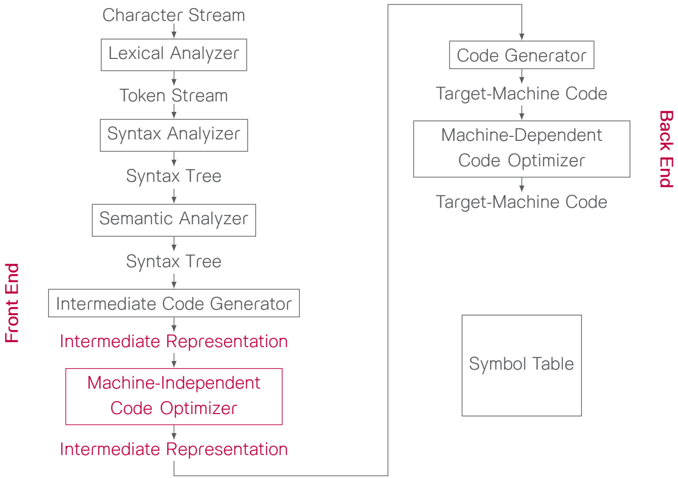
- The optimization **MUST** be correct
- The optimization must improve the performance of many programs
- The compilation time must be kept reasonable
- The engineering effort required must be manageable

By studying compilers, we learn

- The general methodology of solving complex and open-ended problems
- A good example of a software development process

Outline

Phases of a compiler

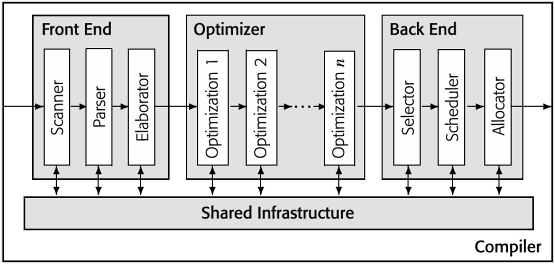


Compiler structure

1. The **front end** must encode its knowledge of the source program in some structures, intermediate representation (IR), for later use
2. The **back end** must map the IR into the instruction set and the finite resources of the target machine

A compiler can make multiple *pasess* over the IR form of the codes and store derived knowledge in the output IR. IR can be a graph, directed graph, or linear code-like forms.

Internal structure of a compiler



■ **FIGURE 1.1** Internal Structure of a Typical Compiler.

Figure: Structure of a compiler[2]

[2]Fig 1.1 from Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022.

Example

We want to translate the following codes

$$a \leftarrow a \times 2 \times b \times c \times d$$

- a, b, c, d are variables
- $\leftarrow$ denotes assignment
- $\times$ is the multiplication operator

The result should be executable codes (machine codes) for the equation

Outline

1. Course Outline
2. Introduction
3. The science of building a compiler
4. The structure of a compiler
   - The front end
   - The back end

The front end

The objective of the front end is to generate IR from the input source. The input must be correct according to the language syntax

In English, we may have a simple grammar

- *Sentence* $\rightarrow$ *Subject* `verb` *Object* `endmark`
- `verb` and `endmark` are part of speech
- *Sentence*, *Subject* and *Object* are syntactic variables
- The symbol "$\rightarrow$" reads ***derives*** and means that the instance of the right-hand side can be abstracted to the syntactic variable on the left-hand side.

The front end process (1)

“Compilers are engineered objects.”

The compiler reads the stream of characters and split them into words and find the corresponding part-of-speech of each word

- (`noun`, “Compiler”), (`verb`, “are”), (`adjective`, “engineered”), (`noun`, “objects”), (`endmark`, “.”)
- The tool in charge is called a **scanner or lexical analyzer**

The front end process (2)

“Compilers are engineered objects.”

The compiler tries to match the sequence of part-of-speech with the grammar

- `noun verb adjective noun endmark`
- The tool in charge is called a **parser**

Some rules in the grammar

1. *Sentence* → *Subject* `verb` *Object* `endmark`
2. *Subject* → `noun`
3. *Object* → *Modifier* `noun`
4. *Modifier* → `adjective`

Rules	Prototype Sentence
-	*Sentence*
1	*Subject* `verb` *Object* `endmark`
2	`noun verb` *Object* `endmark`
3	`noun verb` *Modifier* `noun endmark`

The front end process (3)

- A syntactically correct sentence may be meaningless or have incorrect sense
  - E.g. Girls are good boys.
- The semantic checking will confirm the consistency of meaning in the input
  - In a programming language, semantic may be in the form of data type
  - E.g. a string cannot be multiplied with another string

Intermediate representation

Result from the parsing process and IR optimization

- A graph of execution order
- An assembly-like program

Example: $a \leftarrow a \times 2 \times b \times c \times d$

$$
\begin{aligned}
t_0 &\leftarrow a \times 2 \\
t_1 &\leftarrow t_0 \times b \\
t_2 &\leftarrow t_1 \times c \\
t_3 &\leftarrow t_2 \times d \\
a &\leftarrow t_3
\end{aligned}
$$

The optimizer

After we can see the whole program, we may be able to re-arrange and re-write some operations to optimize the running time, memory usage of the source IR

- Data-flow analysis: how does value change during runtime
- Dependence analysis: does this computation depend on certain memory reference location

Optimization based on context

Supposed that b and c do not change throughout the loop (loop invariant)

```
b ← ···
c ← ···
a ← 1
for i = 1 to n
    read d
    a ← a × 2 × b × c × d
    end
```

(a) Original Code in Context

```
b ← ···
c ← ···
a ← 1
t ← 2 × b × c
for i = 1 to n
    read d
    a ← a × d × t
    end
```

(b) Improved Code

■ **FIGURE 1.3** Context Makes a Difference.

Figure: Optimized code in context[3]

[3] Fig 1.3 from Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022.

Outline

The back end

1. Instruction selection
2. Instruction scheduling
3. Register allocation

We will use ILOC as the target machine code in this course

ILOC

Intermediate Language for an Optimizing Compiler

- A machine-independent assembly-like language

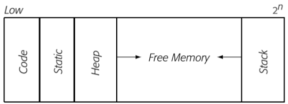
loadAI	r_{arp}, @a	$\Rightarrow$	r_a	// load 'a'
loadI	2	$\Rightarrow$	r_2	// constant 2 into r_2
loadAI	r_{arp}, @b	$\Rightarrow$	r_b	// load 'b'
loadAI	r_{arp}, @c	$\Rightarrow$	r_c	// load 'c'
loadAI	r_{arp}, @d	$\Rightarrow$	r_d	// load 'd'
mult	r_a, r_2	$\Rightarrow$	r_a	// $r_a \leftarrow a \times 2$
mult	r_a, r_b	$\Rightarrow$	r_a	// $r_a \leftarrow (a \times 2) \times b$
mult	r_a, r_c	$\Rightarrow$	r_a	// $r_a \leftarrow (a \times 2 \times b) \times c$
mult	r_a, r_d	$\Rightarrow$	r_a	// $r_a \leftarrow (a \times 2 \times b \times c) \times d$
storeAI	r_a	$\Rightarrow$	r_{arp}, @a	// write r_a back to 'a'

■ **FIGURE 1.4** Example in ILOC.

Figure: ILOC example[4]

[4] Fig 1.4 from Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022.

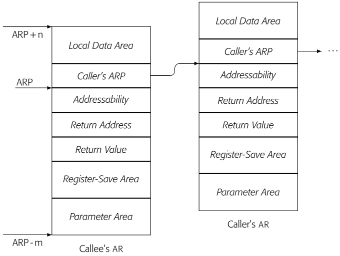
Memory allocation



■ FIGURE 5.14 Virtual Address-Space Layout.

Figure: Memory allocation[a]

[a]Fig 5.14 from Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022.



■ FIGURE 6.2 Typical Activation Records.

Figure: Acitvation record[a]

[a]Fig 6.2 from Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022.

Instruction selection

Machine-specific instructions may improve the efficiency of the system, e.g.

- Immediate-multiply operation will save the cost to load a register with the immediate
  - `multI` vs. `load` and `mult`
- Addition is usually cheaper than multiplication. Therefore, we may re-write `x * 2` with `x + x`

Register Allocation

- In the generation process, we assume that we have unlimited number of register
- But, normally, we have at most a hundred of registers
- We need to free some registers by storing them into the memory (additional load/store instructions)
- We try to save frequently used values in the register to save the unnecessary load/store

loadAI	r_{arp}, @a	$\Rightarrow r_1$	// load 'a'
add	r_1, r_1	$\Rightarrow r_1$	// $r_1 \leftarrow$ a $\times$ 2
loadAI	r_{arp}, @b	$\Rightarrow r_2$	// load 'b'
mult	r_1, r_2	$\Rightarrow r_1$	// $r_1 \leftarrow$ (a $\times$ 2) $\times$ b
loadAI	r_{arp}, @c	$\Rightarrow r_2$	// load 'c'
mult	r_1, r_2	$\Rightarrow r_1$	// $r_1 \leftarrow$ (a $\times$ 2 $\times$ b) $\times$ c
loadAI	r_{arp}, @d	$\Rightarrow r_2$	// load 'd'
mult	r_1, r_2	$\Rightarrow r_1$	// $r_1 \leftarrow$ (a $\times$ 2 $\times$ b $\times$ c) $\times$ d
storeAI	r_1	$\Rightarrow r_{arp}$, @a	// write r_1 back to 'a'

Figure: Register allocation example[5]

Memory instructions require more time than computation instruction

- We must wait until all of the dependent code finished execution
- However, we may load/store values in parallel to the computation if the values are not required by the current computation

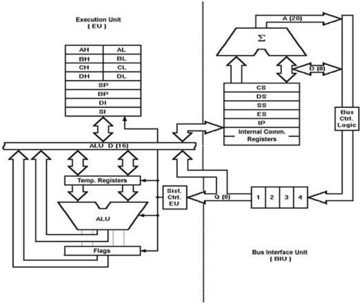


Figure: Execution pipeline

Ex. Instruction scheduling

Start	End	Code
1	3	loadAI r_{arp}, @a ⇒ r_1 // load 'a'
4	4	add r_1, r_1 ⇒ r_1 // $r_1 \leftarrow$ a × 2
5	7	loadAI r_{arp}, @b ⇒ r_2 // load 'b'
8	9	mult r_1, r_2 ⇒ r_1 // $r_1 \leftarrow$ (a × 2) × b
10	12	loadAI r_{arp}, @c ⇒ r_2 // load 'c'
13	14	mult r_1, r_2 ⇒ r_1 // $r_1 \leftarrow$ (a × 2 × b) × c
15	17	loadAI r_{arp}, @d ⇒ r_2 // load 'd'
18	19	mult r_1, r_2 ⇒ r_1 // $r_1 \leftarrow$ (a × 2 × b × c) × d
20	22	storeAI r_1 ⇒ r_{arp}, @a // write r_1 back to 'a'

Figure: Before optimization[a]

[a] Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022., Page 21.

Start	End	Code
1	3	loadAI r_{arp}, @a ⇒ r_1 // load 'a'
2	4	loadAI r_{arp}, @b ⇒ r_2 // load 'b'
3	5	loadAI r_{arp}, @c ⇒ r_3 // load 'c'
4	4	add r_1, r_1 ⇒ r_1 // $r_1 \leftarrow$ a × 2
5	6	mult r_1, r_2 ⇒ r_1 // $r_1 \leftarrow$ (a × 2) × b
6	8	loadAI r_{arp}, @d ⇒ r_2 // load 'd'
7	8	mult r_1, r_3 ⇒ r_1 // $r_1 \leftarrow$ (a × 2 × b) × c
9	10	mult r_1, r_2 ⇒ r_1 // $r_1 \leftarrow$ (a × 2 × b × c) × d
11	13	storeAI r_1 ⇒ r_{arp}, @a // write r_1 back to 'a'

Figure: After optimization[a]

[a] Keith D. Cooper and Linda Torczon. **Engineering a Compiler (Third Edition)**, Morgan Kaufmann, 2022., Page 22.