

Compiler Construction

Lecture 1: Introduction

K.R. Chowdhary, Professor of CS

July 11, 2019 | Fall 2019

Objectives and Outcomes

Objectives

- Basic principles of compiler design, its parts, algorithms and data structures
- Link between computer architecture and compiler design
- Constructing efficient algorithms for compilers

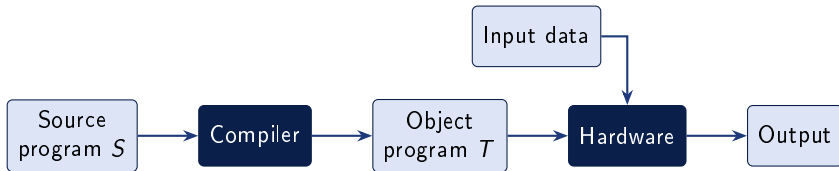
Outcomes

- Know how a compiler works and how source/target languages shape its design
- Lexical analysis, parsing, type-checking, IRs, optimization, register allocation, code generation
- Write, maintain and evolve complex software: correctness, efficiency, modularity

Why study compilers?

Programs must be translated into the machine language of the target hardware. Compiler principles also apply to NLP and information retrieval.

Compiler vs. Interpreter



Compiler

- Translates the entire program first, then runs it
- Faster execution
- Needs error-free translation to produce output
- C, C++, Fortran

Interpreter

- Executes statements one at a time, no translation step
- Performance penalty
- Runs until an error or program end
- Prolog, LISP, Smalltalk

Java: source → byte code → run by the JVM (a program, not hardware).

Compiler Structure and Phases



1. **Lexical analysis:** characters $\rightarrow$ tokens
 2. **Syntax analysis:** tokens $\rightarrow$ syntax tree
 3. **Semantic analysis:** type checking, coercion
 4. **Intermediate code generation:** three-address code
 5. **Code optimization:** faster, smaller code
 6. **Code generation:** assembly / machine code, register assignment
- The **symbol table** is shared by all phases
 - Front end is machine independent; back end is tuned to the target
 - Varying front ends with different back ends yields different compilers

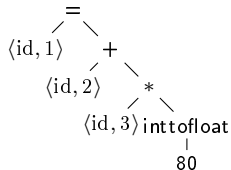
Running Example: $\text{dist} = \text{initdist} + \text{hours} * 80;$

1. Tokens (lexical analysis)

$\langle \text{id}, 1 \rangle \langle = \rangle \langle \text{id}, 2 \rangle \langle + \rangle \langle \text{id}, 3 \rangle \langle * \rangle \langle 80 \rangle$

id entries 1,2,3 point to dist, initdist, hours in the symbol table.

2. Syntax and semantic analysis



3. Intermediate code

```
t1 = inttofloat(80)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

4. Optimized

```
t1 = id3 * 80.0
id1 = id2 + t1
```

5. Target code

```
LD R1, id3
MULT R1, R1, #80.0
LD R2, id2
ADD R1, R1, R2
ST id1, R1
```

Optimization example (loop-invariant code motion)

Original	Optimized
<pre>w = 1 for i = 1 to n read z w = 2*w*x*y*z end</pre>	<pre>w = 1 t = 2*w*x*y for i = 1 to n read z w = w*t*z end</pre>

Goals: faster execution, smaller code, lower memory and power use.

Phase vs. pass

A **phase** is a logical step. A **pass** reads an input file and writes an output file, and may combine several phases.

- Front end phases (lexical, syntax, semantic, IR) typically form one pass
- Back end is one pass per target machine
- Few passes (2–3) are preferred

Key takeaways

- A compiler maps source program S to object program T via analysis and synthesis
- Phases: lexical $\rightarrow$ syntax $\rightarrow$ semantic $\rightarrow$ IR $\rightarrow$ optimizer $\rightarrow$ code generator
- Symbol table connects all phases
- Interpreters execute directly; Java mixes both approaches

Review questions

1. Which phase performs type checking?
2. Why convert integer 80 to float before multiplication?
3. Phase vs. pass?
4. What is a cross-compiler?
5. Why are too many passes undesirable?

Reference: Aho, Lam, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed.