What is a Compiler?

A compiler translates a source specification into a target specification. Traditionally, we consider compilers that take a source language and produce target (machine) code. However, there can be many different types of targets.

<table>
<thead>
<tr>
<th>Source Language</th>
<th>Target Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td>Machine code</td>
</tr>
<tr>
<td>Java</td>
<td>Java Bytecode</td>
</tr>
<tr>
<td>Perl</td>
<td>Perl Bytecode</td>
</tr>
<tr>
<td>Java Bytecode</td>
<td>Machine code</td>
</tr>
</tbody>
</table>

Compilers vs. Interpreters

Compilation – To translate a source program in one language into an executable program in another language and produce results while executing the new program
- Examples: C, C++, FORTRAN

Interpretation – To read a source program and produce the results while understanding that program
- Examples: BASIC, LISP

Hybrid – Try to use both (such as in Java)
1. Translate source code to bytecode
2. Execute by interpretation on a JVM
   or
2. Execute by compilation using a JIT

Java Compiler

Compilation

Pros:
- Fast execution
- Can exploit machine architecture features

Cons:
- Complexity
- Must be done before execution
**Interpreter**

- **Pros:**
  - Machine independent
  - Easy to debug
  - Flexible to modify

- **Cons:**
  - Time overhead
  - Space overhead

---

**Phases of Compilation**

- **Source Code**
  - Lexical Analyzer
    - Token Sequence
  - Syntax Analyzer
    - Syntax Tree
  - Semantic Analyzer
    - Intermediate Representation
  - Code Optimizer
    - Optimized IR
  - Code Generator
    - Assembly/Machine Code

---

**Phases**

**Lexical Analysis**
- Recognize token – smallest stand-alone unit of meaningful information
- Analyze input (strings of characters) from source
  - Scan from left to right
  - Report errors

**Syntax Analysis**
- Group tokens into hierarchical groups
  - Differentiate if-statement, while-statement, ...
  - Report errors

**Semantic Analysis**
- Determine the meaning using the structure
  - Checks are performed to ensure components fit together meaningfully
  - Limited analysis to catch inconsistencies, e.g., type checking
  - Put semantic meaningful items in the structure
  - Produce IR (easier to generate optimized machine code from IRs)

---

**Lexing**

**Input:** Source program
**Output:** Sequence of tokens

**Example:**

```plaintext
if(x > 3)
    y++;
```

```
IF (ID('x') > NUM('3')) { ID('y') INCREMENT };
```

---

**Parsing**

**Input:** Sequence of tokens
**Output:** Abstract Syntax Tree

**Example:**

```
if-stmt
    cond-exp
    stmt-list
    post-inc
```

---
Code Generation

**Input:** Intermediate representation

**Output:** Target code

**Example:**

```
slti $t1, 3, $s0
beq $t1, $zero, L1
addi $s1, $s1, 1
L1:
```

Data Structures for Compilation

**Abstract Syntax Tree**
- Stores the information from the parse and lexing phases
- Walk the tree to produce IR or target code

**Symbol Table**
- Collect and maintain information about identifiers
  - Attributes: type, address, scope, size
- Used by most compiler passes and phases
  - Some phases add information:
    - lexing, parsing, semantic analysis
  - Some phases use information:
    - Semantic analysis, code optimization, code generation
- Debuggers also can make use of a symbol table
  - `gcc -g` keeps a version of the symbol table in the object code

Three-pass Compiler

**Passes:** number of times through a program representation
- 1-pass, 2-pass, multi-pass compilation
- Language becomes more complex → more passes

**Phases:** conceptual and sometimes physical stages
- Symbol table coordinates information between phases
- Phases are not completely separate
- Semantic phase may do things that syntax phase should do
- Interaction is possible

Compiler Construction

**Automatic Generators:**
- Lexical Analysis — Lex, Flex, JLex, JFlex
- Syntax Analysis — Yacc, Bison, JavaCUP, JavaCC
- Semantic Analysis
- Code Optimization
- Code Generation