Chapter 1. Introduction

I. Basics
   a.) Terminology
   b.) Compiler vs. Interpreter

II. Compilation Process
   a.) Phases of compilation process
   b.) Supporting systems
   c.) pass

III. Writing a compiler
   a.) Choice of a language
   b.) Bootstrapping
   c.) Re-targetable compilers

I. Basics
   a.)
   Q: What is a compiler?
   A: Software that translates a code written in a source language, such as c, c++, … into a code of an object language, normally some kind of assembly.
   Cross compiler: A compiler that can generate code to run on different kind of machine.
   Run-time library (Book page 5, Figure 1.1)
   b.) Compiler vs. Interpreter
difference: Interpreter does not create object code. It executes source code line by line. Interpreter is good for debugging. But it is slower. E.g. LISP, traditional BASIC, JAVA…
II Compilation process
   a.) 5 step process

   Source Code \rightarrow Lexical Analysis (Ch2) \rightarrow Syntax Analysis (Ch 3, 4) \rightarrow Intermediate Code Generation (Ch 5) \rightarrow Optimization (Ch 6) \rightarrow Object Code Generation (Ch 7) \rightarrow Object Code

   Symbol Table (Ch 8)

Appendix A. How to minimize the number of states of a DFSM
Appendix B. Lex and Yacc
1. **Lexical Analysis**: it is a phase where the source code is broken up into “meaningful units” called tokens.

   Example: source: if (x==y) a = b – 5;
   - Tokens: if, (, x, ==, y, ), a, =, b, -, 5, ;.
   - Ignore comments and white spaces
   - case conversion if needed.
   - Preprocessor; interprets compiler directives.

2. **Syntax Analysis**: it is a phase where the structure of a program/ individual construct is recognized and constructed.

   Example: program ➔ functions, procedures, types, …
   Individual Construct ➔ statements, expressions, operations, …

   Example: Grammar <define a language>
   An assignment statement.

   Production rules:
   - `<statement>` ➔ `<identifier>` = `<expression>`,
   - `<expression>` ➔ `<expression>` + `<expression>`,
   - `<expression>` ➔ `<expression>` – `<expression>`,
   - `<expression>` ➔ `<identifier>`

   source: a=b+c;

   ![Diagram of expression tree]

3. **Intermediate Code Generation**: is a phase where an internal representation of the source is created that reflects the information uncovered during syntax Analysis.

   Intermediate Code:
   (1) AST (Abstract Syntax Tree)
   Remove unnecessary information from parse tree.
   AST = PST – “unnecessary info”

   Example:
(2) 3AC (Three Address Code)
An instruction with at most 3 addresses (variables) and one operation.
Example: \( a = b+c; \)
\[ T_1 = b+c; \]
\[ a = T_1; \]
Example: \( x = a+b*(c - d); \)
1. \( T_1 = c - d \)
2. \( T_2 = b * T_1 \)
3. \( T_3 = a + T_2 \)
4. \( X = T_3 \)

4. Optimization: it is a process of identifying and removing unnecessary and/or redundant operations from IC (intermediate code).
Example:
Combine 3 and 4. \( x = a+T_2 \)
2 kinds of optimizations:
Machine independent optimization tries to optimize the intermediate code.
Machine dependent optimization tries to optimize the object code.

5. Object Code Generation: it is a phase where the intermediate code is translated into an object code.
Example:
\[
\begin{align*}
L 2, c \\
S 2, d \\
M 2, b \\
A 2, a \\
STO 2, x
\end{align*}
\]

b.) supporting systems
(1) Symbol Table handler (Chpt 8)
Depository for all information during the compilation
2 operations: Insert and look-up.
(2) Error Handler (Chpt 5)
Gives meaningful error message (line #, token found; syntax error, …)
Recovery from the error
c.)pass: it is a reading one version of a program from a file and writing a new version to a file.
Example: 1-pass compiler (everything needs to be defined before being used)
2-pass compiler
multiple-pass compiler

III. Writing a compiler
a.) Compiler for JOHN (new language)
In C. \( \rightarrow \) C compiler \( \rightarrow \) compiler for JOHN.
Q: Can we write a compiler in its own language?
Yes/No.
b.) Bootstrapping
Minimal compiler for JOHN in c → C compiler → minimal compiler for JOHN;
→ (multiple steps) → Full/complete compiler for JOHN in minimal JOHN →
minimal JOHN compiler → Full JOHN compiler

c.) Re-targetable Compilers

2 ways to build such a compiler
1. distinguish front- and back-end of a compiler.
   Front-end: From Lexical Analysis to machine-independent optimization.
   Back-end: From machine-dependent optimization to object code generation.

   Build one Front-end and attach it to different versions of back-ends to
   generate different object code for different machines.

   ![Diagram of re-targetable compilers](image)

2. Creating an object code for a “virtual machine” → “P-code”

   Source → Compiler for P-code → P-code → Translator1 → Object Code 1
   → Translator2 → Object Code 2
   → Translator3 → Object Code 3