Section: The Structure of a Compiler

1.1 What is a Compiler?

I. Translator

Definition:

\[
\begin{array}{c}
\text{program in} \\
\text{language} \\
X
\end{array} \quad \xrightarrow{\text{translator}} \quad \\
\begin{array}{c}
\text{for} \\
X
\end{array} \quad \xrightarrow{} \quad \\
\begin{array}{c}
\text{program in} \\
\text{language} \\
Y
\end{array}
\]
Examples:

<table>
<thead>
<tr>
<th>Source Language</th>
<th>Object Language</th>
<th>Name</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>High</td>
<td>preproc</td>
<td>ratfor $\rightarrow$ f77</td>
</tr>
<tr>
<td>Assem.</td>
<td>Mach.</td>
<td>assemb</td>
<td>as</td>
</tr>
<tr>
<td>High</td>
<td>Mach.</td>
<td>compil</td>
<td>g++, javac</td>
</tr>
<tr>
<td>Any Level</td>
<td>executes</td>
<td>interp</td>
<td>BASIC c shell</td>
</tr>
<tr>
<td></td>
<td>immed.</td>
<td></td>
<td>apl, lisp</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>java</td>
</tr>
</tbody>
</table>
• Preprocessor

\[
\text{for } i=1 \text{ to } n \text{ do } \\
\text{ (stmts) } \\
\text{ end for}
\]

\[
\downarrow
\]

\[
i = 1 \\
\text{while } (i <= n) \text{ do } \\
\text{ (stmts) } \\
i = i + 1 \\
\text{ end while}
\]
skeletal source program
↓

preprocessor
↓

source program
↓

compiler
↓

target (object) assembly program
↓

assembler
↓

relocatable machine code
↓

loader/link-editor
↓

absolute machine code
III. Compiler

program in high level language X $\rightarrow$ compiler for X $\rightarrow$ program in machine language Y
1.2 Structure of a Compiler

General Overview

Source Code

→

Lexical Analysis

→

tokens

Syntax Analysis

→

parse trees

Symbol Table Management

Intermediate Code Generation

→

intermediate code

Error Handling

→

intermediate code

Code Optimization

→

Code Generation

→

Object Program
1.3 PHASES OF COMPILATION

1.3.1 Lexical Analysis (Scanner)

a. Purpose: Read the same program character by character grouping them into atomic units called “tokens.”

b. Tokens:

• depend on language and compiler writer
• Examples:

  reserved words    if, for
  operators        +, −, <, =
  constants        0, 4.89
  punctuation      (, }, [ 
  identifiers      sb, ch

• treated as a pair: token.type and token.value
c. Example

if (x <= 0) x = y + z

when put through lexical analyzer produces:

<table>
<thead>
<tr>
<th>token</th>
<th>type</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>23</td>
<td>“x”</td>
</tr>
<tr>
<td>&lt;=</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>int constant</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>)</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>23</td>
<td>“x”</td>
</tr>
<tr>
<td>= assignment</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>23</td>
<td>“y”</td>
</tr>
<tr>
<td>+</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>23</td>
<td>“z”</td>
</tr>
</tbody>
</table>
d. How does one build a lexical analyzer?

• from scratch
• lex

e. Preview of Lex

• idea: tokens described by regular expressions
• basic syntax: regular expression, action
• basic semantics: if match regular expression, then do action.
• Example:

```markdown
%%
"if" return(25);
"(" return(28);
[0-9]+ return(22);
```

f. Remarks
Besides returning token types and values, the lexical analyzer might

a) print error messages

b) insert identifiers in the symbol table

1.3.2 Syntax Analysis (Parsing)

a. Purpose:

b. Syntax:
c. Parse Tree

\[ \text{if (} x \leq 0 \text{) } x = y + z \]

![Parse Tree Diagram]

d. How does one build a parser?

- from scratch
- using a parser generator such as yacc
1.3.3 Intermediate Code Generator

a. Purpose: Traverse the parse tree, producing simple intermediate code.

b. Three-Address Code:

Instructions:

1. id := id op id
2. goto label
3. if condition goto label
Example:

\[
\text{if } (x \leq 0) \quad x = x + z
\]

\[
\downarrow
\]

\[
\text{if } (x \leq 0) \quad \text{goto L1} \\
\text{goto L2}
\]

L1: \quad x := y + z

L2:

1.3.4 Intermediate Code Generation
a. Purpose: Transform the intermediate code into “better” code.
b. Examples

1) Rearrangement of Code

if (x <= 0) goto L1
if (x > 0) goto L2

goto L2

→ \[ x = y + z \]

L1: \[ x = y + z \]
L2:

2) Redundancy Elimination

\[ a = w + x + y \]
\[ T1 = x + y \]

→ \[ a = w + T1 \]

\[ b = x + y + z \]
\[ b = T1 + z \]
3) Strength Reduction

\[ x^2 \rightarrow x \times x \]

expensive \rightarrow cheap
operator \rightarrow operator

4) Frequency Reduction

\begin{verbatim}
for (i=1; i<n; i=i+1) {
    T1 = sqrt(26)
    x = sqrt(26)
    for (i=1; i<n; i=i+1) {
        x = T1
    }
}
\end{verbatim}

c. Remarks:

1) Main criteria for optimization is speed.
1.3.5 Code Generation

a. Purpose: Transform intermediate code to machine code (assembler)

b. Example: $a = b + c$

```
mov  b, R1
add  c, R1
mov  R1, a
```

c. Remarks
1.4 Symbol Table

a. Purpose: record information about various objects in the source program

b. Examples

- procedure - no. and type of arguments
- simple variable - type
- array - type, size

c. Use - information is required during

- parsing
- code generation
1.5 Error Handler

a. Errors - all errors should be

  - detected
  - detected correctly
  - detected as soon as possible
  - reported at the appropriate place and in a helpful manner

b. Purpose

  - report errors
  - “error recovery” - proceed with processing
c. Note: Errors can occur in each phase

- misspelled token
- wrong syntax
- improper procedure call
- statements that cannot be reached