1.1 What is a Compiler?

I. Translator

Definition:

```
program in translator program in
language X for language X
```

Examples:

<table>
<thead>
<tr>
<th>Source Language</th>
<th>Object Language</th>
<th>Name</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>High Level</td>
<td>High Level</td>
<td>preprocessor</td>
<td>ratfor → f77</td>
</tr>
<tr>
<td>Assembly</td>
<td>Machine</td>
<td>assembler</td>
<td>as</td>
</tr>
<tr>
<td>High Level</td>
<td>Machine</td>
<td>compiler</td>
<td>f77, cc, pc</td>
</tr>
<tr>
<td>Any</td>
<td>executes</td>
<td>interpreter</td>
<td>BASIC (often)</td>
</tr>
<tr>
<td></td>
<td>immediately</td>
<td></td>
<td>c shell</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>apl, lisp</td>
</tr>
</tbody>
</table>

- Preprocessor

```plaintext
for i=1 to n do
    (stmts)
end for
```

↓

```
i = 1
while (i<=n) do
    (stmts)
    i = i + 1
end while
```
II. Language Processing System

skeletal source program

\[ \downarrow \]

preprocessor

\[ \downarrow \]

source program

\[ \downarrow \]

compiler

\[ \downarrow \]

target (object) assembly program

\[ \downarrow \]

assembler

\[ \downarrow \]

relocatable machine code

\[ \downarrow \]

loader/link-editor

\[ \downarrow \]

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

↓ intermediate code

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
  - token type is a (mnemonic) integer
  - some tokens have no 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:
```
%%
"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: Accepts the sequence of tokens generated by the lexical analyzer, checks whether the program is syntactically correct, and generates a parse tree.

b. Syntax: formally described by a context free grammar.
c. Parse Tree

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

Figure 2 is the parse tree for this statement.

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. \( \text{id} := \text{id} \text{ op } \text{id} \)
2. \( \text{goto label} \)
3. \( \text{if condition goto label} \)
Example:

```plaintext
if (x<0) x = x + z
↓
if (x<0) goto L1
goto L2
L1: 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

```plaintext
if (x<0) goto L1  if (x>0 goto L2
goto L2 → x = y + z
L1: x = y + z  L2:
L2:
```

2) Redundancy Elimination

```plaintext
a = w + x + y → T1 = x + y
b = x + y + z → a = w + T1
b = T1 + z
```

3) Strength Reduction

```plaintext
x^2 → x * x
expensive → cheap
operator  operator
```

4) Frequency Reduction

```plaintext
for (i=1; i<n; i=i+1) {
x = sqrt(26)
} → T1 = sqrt(26)
for (i=1; i<n; i=i+1) {
x = T1
}
```
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: \( a = b + c \)

\[
\begin{align*}
\text{mov} & \quad b, \text{R1} \\
\text{add} & \quad c, \text{R1} \\
\text{mov} & \quad \text{R1}, a
\end{align*}
\]

c. Remarks

1) completely machine dependent whereas other phases are not

2) “register allocation” is the most difficult task

- idea - use registers (fast access) to avoid memory use (slow access)
- problem - only a finite number of registers (during intermediate code phase, one assumes an infinite number)

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