Chapter 3.2

Functions and Purposes of Translators

3.2 (a) Interpreters and Compilers

- All computer programs are written in programming languages.
- Programming languages are of three types
  - Low Level (Machine Language)
  - Mid Level (Assembly Language)
  - High Level (C, BASIC, COBOL, FORTRAN etc.)
- Program written in Mid or High Level language is called Source Code.
- Processors only understand Machine Language.
- Source Code must be "translated" into Machine Language, before it can be run.
- A special program called a "translator" is used for this conversion.
- The converted code is called the "Object Code".

3.2 (a) Interpreters and Compilers

- **Translators are divided into two categories:**
  - **Interpreters**
  - **Compilers**
- **Underlying conversion concepts are almost similar.**

3.2 (a) Interpreters and Compilers

Interpreters

- **Interpreter reads the source code line by line, starting with the first line.**

- **Interpreter keeps translating and executing the source code line by line until the code is completely run, or a syntax error is encountered, in which case the interpreter will stop.**

- **An interpreter does not create an object file. It decodes and runs each line without saving it to an object file.**

3.2 (a) Interpreters and Compilers

Compilers

•Compiler reads the entire source code before translating it into machine language.

•Compiler will not translate the source code if it finds a single syntax error in it.

•Once the code is found error-free, the compiler will produce the machine language file (object code), which can be run directly by the OS.

3.2 (a) Interpreters and Compilers

Merits and Demerits of Interpreters

Merits	Demerits
•Interpreter is a relatively simple piece of software and hence is easier to write. •Interpreter doesn't need to load the entire source code into the computer's memory and hence can be run on machines with low system resources. Early translators were all interpreters. •Interpreter starts execution immediately by translating and executing the first line of source code. This reduces execution time while the source code is being developed.	•Interpreter is slow on subsequent executions as it translates the source code each time. •As interpreter doesn't create an object file the source code must be distributed along with the interpreter in order for the user to run the software. •Interpreter has to decode lines that have already been decoded and executed before (e.g. lines in a loop).

3.2 (a) Interpreters and Compilers

Merits and Demerits of Compilers

Merits	Demerits
•Compiler reads the entire source code before translating it so it can catch any syntax error before the execution of the code begins. •Once compiled, the object code doesn't need to be translated again and again, unless the source code is changed. This will speed up subsequent executions. •All commercial software (non open source) is compiled, as the source code is copy righted. The user only gets the object code which can be directly run on his/her machine.	•Compiler is a complex software as it needs to perform many complicated tasks before translating the source code. Hence, it is more difficult to write a compiler. •Compiler needs to load itself and the entire source code into computer's memory in order to translate it. This means that a compiler requires more system resources. •For first-time execution, a compiler is slow to start as it must read the entire source code before it can begin to execute it.

3.2 (a) Interpreters and Compilers

Keywords

1. Translator
2. source code
3. syntax error
4. object code
5. Compiler
6. Interpreter
7. conversion
8. machine code
9. Assembly language
10. Assembler
11. high-level language
12. low-level language

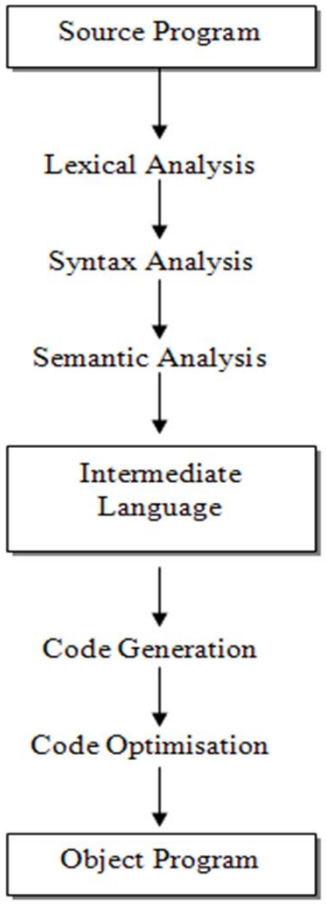
3.2 (b) Lexical Analysis

The Translation Process

- **The basic translation process is identical for both interpreters and compilers.**

- **Source code goes through various stages and is converted into Intermediate Language.**

- **From this, final object code is generated, which can be optimized.**



3.2 (b) Lexical Analysis

The Translation Process

1. **White spaces, blank lines and comments are removed from the code.**

2. **Using the grammar of the language being used, the lexical analyzer assigns tokens to a meaningful string of characters. Single characters are converted into their ASCII codes.**

3. **A token could be anything from 16-bit unsigned integers (starting from 256) to simple labels.**

4. **Variable names require extra information. A symbol table is used to keep record of variables. This table is used throughout the translation process. During Lexical Analysis, only variable names are noted into the symbol table.**

5. **The symbol table is stored as a Linked List and searching is performed using hashing.**

6. **Some basic error reporting is done in this stage. E.g. Illegal Identifier…**

3.2 (b) Lexical Analysis

Example of Tokenization

C Language statement: sum = 3 + 2;

lexeme	token type
sum	IDENTIFIER
=	ASSIGN_OP
3	NUMBER
+	ADD_OP
2	NUMBER
;	SEMICOLON

3.2 (b) Lexical Analysis

When Lexical Analysis phase is finished, the code is converted into a standard format that the Syntax Analyzer can understand.

Keywords:

1. White spaces
2. Comments
3. Tokens
4. Lexeme
5. 16-bit unsigned integers
6. Labels
7. Symbol Table
8. Illegal Identifier error
9. Linked List
10. Hashing
11. Standard Format

3.2 (c) Syntax Analysis

- All computer languages have their specific grammar (syntax of writing valid programming statements.
- This grammar is defined using BNF (Backus-Naur Form).
- The Syntax Analyzer (or Parser) will analyze the tokenized code against the grammar of the language.
- The parsing transforms the code into data structure, usually a Binary Tree, which is suitable for further processing.
- Invalid command names, such as INPT instead INPUT will be identified at this point.
- Some languages require variables to be declared before they can be used. The syntax analyzer will catch variables without declarations (using the symbol table).

3.2 (c) Syntax Analysis

Example of Parsing Using BNF

Assignment Statement ::= <variable> <assignment operator> <expression>

Expression ::= <variable> <arithmetic operator> <variable>

The statement sum = sum + number will be parsed as

<variable> <assignment operator> <variable> <arithmetic operator> <variable>

<variable> <assignment operator> <expression>

<assignment statement> which is syntactically valid.

If the original statement was sum = sum + + number it will be parsed as

<variable> <assignment operator> <variable> <arithmetic operator> <arithmetic operator> <variable>

which is syntactically invalid.

3.2 (c) Syntax Analysis

Semantic Analysis

In this part of the analysis, the code is checked for flow of control, labels, variable declaration and their types.

For example, any jump statement that requires the program control to jump to a certain point in the code identified by a label must be validated. If no such label exits, and error will be produced.

Similarly, certain programming construct can only be placed in certain parts of the code. For example, the EXIT FOR statement in Microsoft Visual Basic can only be placed inside a FOR…NEXT loop. If the syntax analyzer (parser) finds an EXIT FOR statement outside a FOR…NEXT loop, it generates an error.

Most modern compiler can point of such errors with fair accuracy, but this may not the case each time as the code can be quite complex.

3.2 (c) Syntax Analysis

Keywords:

1. tokenization
2. parser
3. BNF
4. standard format
5. error identification
6. semantic analysis
7. binary tree
8. data structure

3.2 (d) Code Generation

Code Generation

- All errors due to incorrect use of language have been removed at this stage.
- The code is now ready to be translated into a form suitable for processor to run.
- Addresses of variables (mentioned in the Symbol Table) are now calculated and stored in their respective entries in the Symbol Table.
    - This is done as soon as these variables are encountered during code generation.
- An intermediate code called “intermediate language” is produced.
    - Some modern languages (Java and .NET languages) stop at this stage and the actual code is compiled on the target machine when the program is run there. This is called ‘just-in-time’ compiling or JIT compiling.
- The intermediate code is then translated or interpreted into machine code which is an executable form of the source code.
    - If the code is compiled, the resultant file can be saved and run subsequently without any further need of translation.

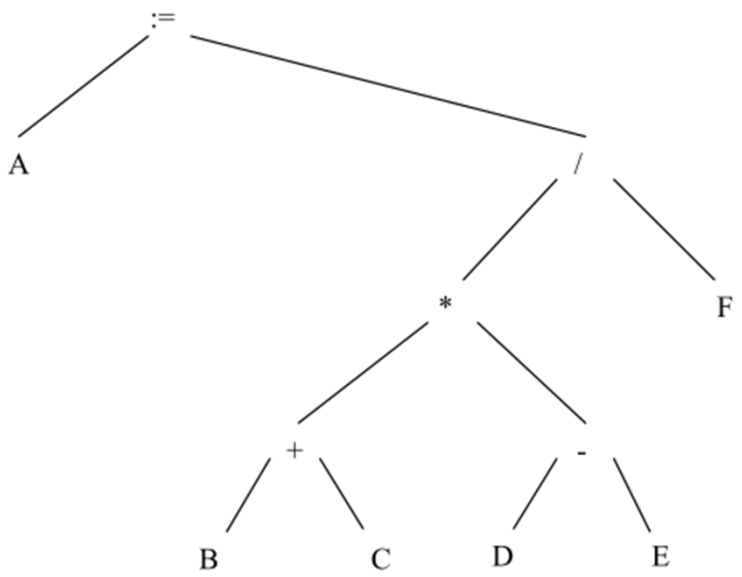
3.2 (d) Code Generation

Code Generation

- Two methods of can be used to represent the High Level language in Machine Code

Example: $A := (B + C) * (D - E) / F$

- Binary Tree Method



3.2 (d) Code Generation

Code Generation

The Two Address Code Method

Example: $A := (B + C) * (D - E) / F$

- Two Address Code (TAC)

$$Operand_1 := Operand_2\ Operator\ Operand_3$$

$$R_1 := B + C$$

$$R_2 := D - E$$

$$R_3 := R_1 * R_2$$

$$A := R_3 / F$$

3.2 (d) Code Generation

Code Optimization

- After code is generated, the compiler may decide what type of optimization can be performed on the generated code either to make the code run faster or reduce the size of the generated code.