Lecture No. 1

Unit-1st

(Introduction)

Compiler Design

(CSEG3015)

Prerequisite/Recapitulations

- Knowledge of automata theory
- Context Free Languages
- Computer Architecture
- Data Structures
- Simple graph algorithms

Objectives

- Translators
- Types of Translators
- Introduction to Compiler
- Structure of Compiler
- Phases and Passes

Introduction

- Translators: is a software which is used to transform or convert from one form (source language) to the another (target language).
- Types of Translators:

a) Compiler

b) Interpreter

c) Assembler

d) Preprocessors

Introduction

a) Compiler: It is a translator, which convert high level language (FORTRAN, PL/I, COBOL) into low level language such as assembly language or machine language.

b) Interpreter: Convert source program to object program line by line.

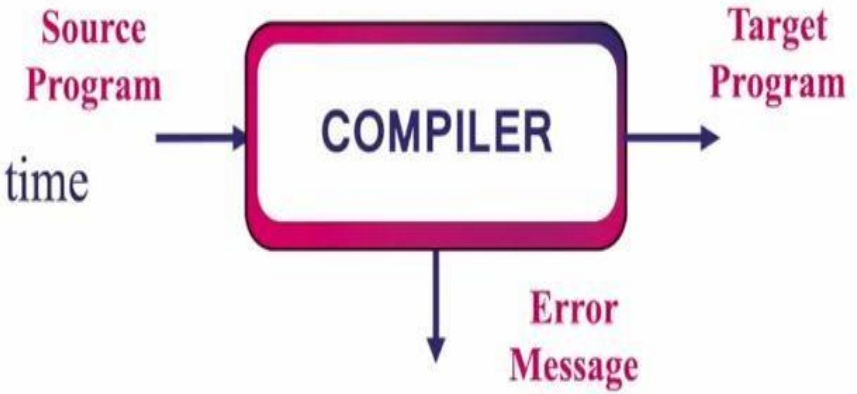
c) Assembler: convert assembly language into machine language.

d) Preprocessor: is sometimes used for translators, that takes program in one high-level language into equivalent program in another high-level language.

COMPILER

- Compiler is a software/program that read the program in one language called source code and translates it into an object code

- **Characteristics**
  - Converts entire source program at a time
  - Detects all syntax errors at a time
  - Requires huge memory space for processing
  - Executes very fast



INTERPRETER

- Interpreter is a software that translates the source program in HLL to object code(machine level language) line by line
- It is the simplest form of translation
- *Characteristics*
  - Line by Line translation
  - Detects only one error at a time
  - Requires less memory space
  - Slow in execution

ASSEMBLER

- Assembler is a software that converts low level/assembly language to machine code
- The object code is meant for specific hardware only, not easily transferable from one file to another file

- *Characteristics*
  - Instructions will be in mnemonics codes
  - Produces object code as output
  - Object code is hardware specific

ASSEMBLER & COMPILER

ASSEMBLER

1. Translates source code in assembly language to machine language
2. Slow execution
3. Less sophisticated program
4. Instructions will be in mnemonic codes
5. The object code need to be linked to run on a machine

COMPILER

1. Translates source code in High level language to machine language
2. Executes very fast
3. More sophisticated program
4. High level programming instructions
5. Compilers produce machine executable code directly from HLL

COMPILER & INTERPRETER

COMPILER

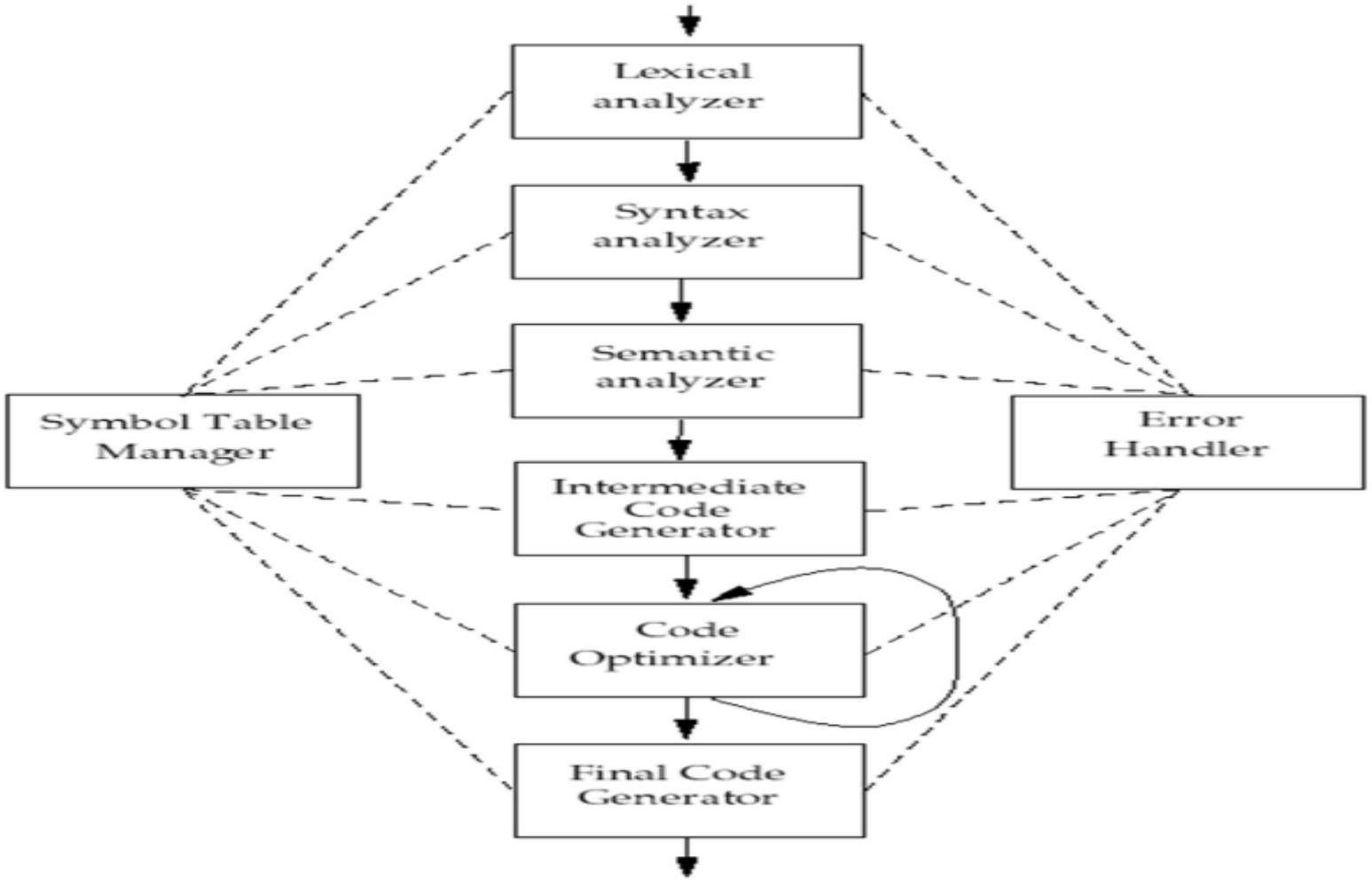
1. Translates complete source code in High level language to machine language
2. Executes very fast
3. More sophisticated program
4. Errors cannot be easily corrected
5. Requires large memory space
6. Suitable for large programs
7. Example: C, C++

INTREPRETER

1. Translates source code to machine language line by line
2. Executes slowly
3. Less sophisticated program
4. Errors can be easily corrected
5. Requires less memory
6. Suitable for small programs
7. Example: Python, Perl

Structure of Compiler

Structure of Compiler:



PHASES/STRUCTURE OF THE COMPILER

Two Main Parts of Compiler

Analysis Part

- Impose Grammatical Structure
- Intermediate Representation
- Maintains Symbol Table
- Front end of the compiler
- **Three Phases**
  - Lexical Analysis
  - Syntax Analysis
  - Semantic Analysis

Synthesis Part

- Desired target program
- Back end of the compiler
- **Three Phases**
  - Intermediate code generation
  - Code Optimization
  - Code Generation

Lexical Analysis Phase

- **Lexical Analysis Phases:** also called scanner, separates characters of the source language into groups that logically belongs together; these groups are called tokens.

➢ **Tokens:** single meaning entity in programming language.

a) Specific Strings(Keywords): can't assign values to these tokens.

b) Classes of strings(Constant, identifiers): can assign values.

➢ Output of lexical analysis is the stream of tokens, which is passed to the next phase.

Lexical Analysis Phase

- Lexical Analysis is the interface between the source program and the compiler.
- Lexical Analysis act as a coroutine and subroutine of syntax analyser.

Syntax Analysis Phase

- **Syntax Analysis Phases:** also called parser, parser has two functions:

a) To check that the token produced by the lexical analyser will match with the specification in language.

b) To make explicit the hierarchical structure of the incoming token streams by identifying which part of the token stream should be grouped together.

c)For example: A/B*C

d)Parse tree for A/B*C -- ????

Semantic Analysis Phase

- **Semantic Analysis Phases:**

1) The term is applied to the def. Of the types of the intermediate results.

2) Semantic Analysis can be done during the syntax analysis phase, the intermediate code generation or the final code generation phase.

3) Whether the output is permitted or not according to machine.

SEMANTIC ANALYSIS

- Compiler checks for **Semantic Consistency/Errors**
- Type checking
- Example
  - Array index
  - Type Mismatch
  - Matching Parenthesis
  - Matching if…else
  - Scope of operation/variable

Intermediate Code Generation

- **Intermediate Code Generation:** Output of the syntax analyser is some representation of the parse tree. ICG phase transform this parse tree into an intermediate language representation of the source program.

➢ For this Three-Address-Code is used.

➢Popular type of intermediate language is Three-Address-Code

➢A=B op C (op is binary operator)

➢T1 = A/B

➢T2 = T1 * C

Code Optimization

- **Code Optimization:** Optional phase of the compiler
- Two types of Optimization:

  1) Local Optimization:

     a) Jump Over Jump

     b) Common Subexpression Elimination

  2) Loop Optimization

  3) Analysis of flow of control and data.

Code Generation

- **Code Generation:** This phase converts the intermediate code into a sequence of machine instructions.

  i.e A:=B+C

  LOAD B

  ADD C

  STORE A

Symbol Table

- **Symbol Table:** It keeps all the information related to the data objects. Example: information of a variable.

➢ Information about data object is collected by the early phase of the compiler – Lexical & Syntactic Analysis – and entered into the symbol table.

➢ Identifier: -- MAX checks whether it is new one or old, if old then update the pointer, otherwise, specified by an identifier in the symbol table.

Error Handler

- **Error Handler:** Types of error in each phase

1) LA: may be unable to proceed because the next token in the source program is misspelled.
2) SA: missing parenthesis has occured.
3) ICG: check the compatibility.
4) CO: remove the code that can never be reached.
5) CG: constant are within the language range or not.
6) Symbol Table: if more then one entry of same variable.

Phases and Passes

- **Phases and Passes:**
- **Phase:-** is a logically cohesive operation that takes as input one representation of the source code and produce as output another representation.
- **Passes:-** refers to the traversal of a compiler through the entire program.

TOKENS, PATTERNS AND LEXEME

- **Lexeme :** Sequence of characters matched with pattern of token
- **Patterns :** Set of rules describes the token
- **Tokens :** Abstract symbol representing lexical unit
- **Example**

```
int x = 20;
```

DESCRIPTION OF TOKENS

Lexeme	Token	Description
if, else, for	Keyword	Reserved words
score, s2, test	Identifier	Any combination of letters and digits
2.765, 0, 4.03e2, 10	Number	Numbers and Constants
+, -, <, > ,>=, <=, !=	Operators	Operators
“Welcome back”	Literal	Anything within Quotes
(, {, , ,; ,: ,’…….	Punctuation Symbol	Parenthesis and Separators

EXAMPLE

```
int add(int a, int b)
//Adds two Numbers
{
        int c;
        c=a+b;
        return c;
}
```

S.No.	Lexeme	Token
1.	int	Keyword
2.	add	Identifier
3.	(	Punctuation
4.	int	Keyword
5.	a	Identifier
6.	,	Punctuation
7.	int	Keyword
8.	b	Identifier
9.	)	Punctuation
10.	{	Punctuation
11.	int	Keyword
12.	c	Identifier

S.No.	Lexeme	Token
13.	;	Punctuation
14.	c	Identifier
15.	=	Operator
16.	a	Identifier
17.	+	Operator
18.	b	Identifier
19.	;	Punctuation
20.	return	Keyword
21.	c	Identifier
22.	;	Punctuation
23.	}	Punctuation

References

- **References:**

Text Books

1. ALFRED V AHO, JEFFREY D ULLMAN “Principles of Compiler Design”.
2. V Raghavan, “ Principles of Compiler Design”, TMH
3. Kenneth Louden,” Compiler Construction”, Cengage Learning

Reference Books

1. Aho, Sethi & Ullman, "Compilers: Principles, Techniques and Tools”,Pearson Education 2
2. Charles Fischer and Ricard LeBlanc,” Crafting a Compiler with C”, Pearson Education

Thank You