Compiler Construction (CS-636)

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk Whatsapp# 0346-5100010

(Week 3) Lecture 5 & 6

Objectives: Learning objectives of these lectures are

Students will able to understand:

- What is Structure of Compiler?
  - Analysis Part (Front-end)
  - Synthesis Part (Back-end)
- What is difference between Pass and Phase?
- What are the roles of Lexical Analyzer?
- What is Token, Pattern & Lexemes?
- What is Input Buffering?
- How to recognize Tokens?
- What are the roles of Syntax Analyzer?
- What is Syntax Error Handling?

Text Book & Resources:

Compilers Principles Techniques and Tools (2nd Edition) by Alfread V. Aho, Ravi Sethi.

Videos Links:

https://youtu.be/X5ZKZrPMDlE (Part 1)
https://youtu.be/tp4fxWWa4hY (Part 2)

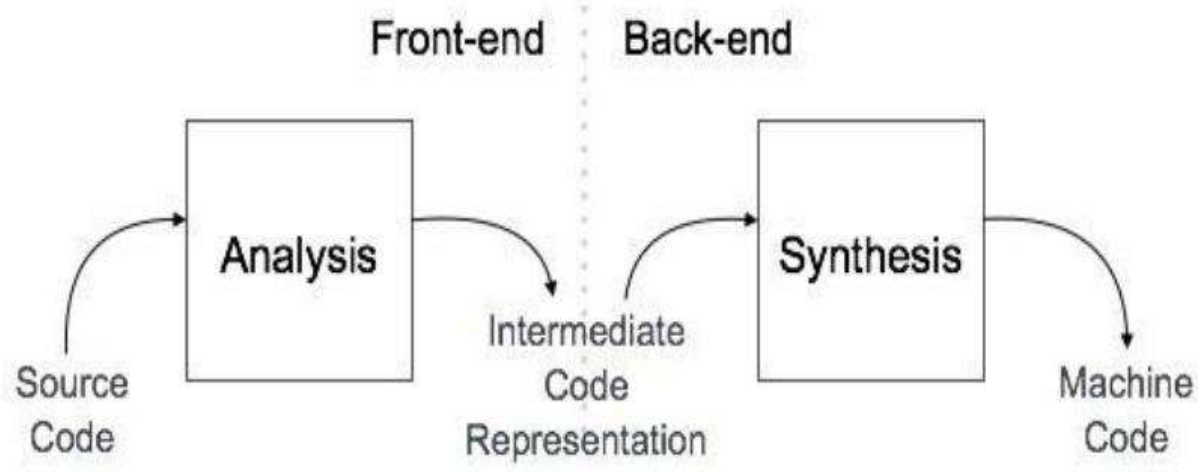
Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk Whatsapp# 0346-5100010

Structure of a Compiler

Up to this point we have treated a compiler as a single box that maps a source program into a semantically equivalent target program. If we open up this box a little, we see that there are two parts to this mapping: **Analysis and Synthesis.**

The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.

It also known as the front-end of the compiler, the analysis phase of the compiler reads the source program, divides it into core parts, and then checks for lexical, grammar, and syntax errors. The analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.



The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table. If we examine the compilation process in more detail, we see that it operates as a sequence of phases, each of which transforms one representation of the source program to another. It is also known as the back-end of the compiler, the synthesis phase generates the target program with the help of intermediate source code representation and symbol table.

A typical decomposition of a compiler into phases is shown in Figure (Week#2 Lectures material). In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly. The symbol table, which stores information about the entire source program, is used by all phases of the compiler.

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk Whatsapp# 0346-5100010

A compiler can have many phases and passes.

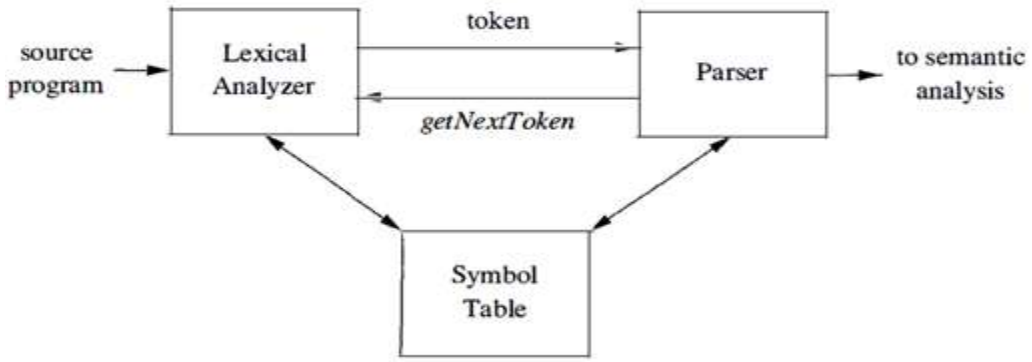
- **Pass:** A pass refers to the traversal of a compiler through the entire program.
- **Phase:** A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage. A pass can have more than one phase.

Some compilers have a machine-independent optimization phase between the front end and the back end. The purpose of this optimization phase is to perform transformations on the intermediate representation, so that the back end can produce a better target program than it would have otherwise produced from an un-optimized intermediate representation. Since optimization is optional, one or the other of the two optimization phases shown in Figure may be missing.

Role of Lexical Analyzer:

- Reads the input characters of source program.
- Group them into lexemes.
- Produce sequence of tokens for each lexeme as output.
- Tokens are sent to the parser for syntax analysis.
- If a lexemes is found as identifier then makes its entry into symbol table.
- LA also collect information from symbol table in some cases.
- Get next token command causes the LA to read character from its input until it can identify the next lexeme and produce for it the next token which it returns to the parser.
- It discards comments, preprocessor directives and white space (blanks, new line, tabs, and perhaps other characters that are used to separate tokens).
- Keep tracks of the numbers of new line character.
- Its correlates error message generated by compiler.
- LA is divided into two processes.
    1. **Scanning**: Do not tokenize the input, delete comments and compact consecutive white space character into one.
    2. **Lexical Analysis:** Produces the sequence of tokens as output.
- Lexical Errors: Lexical Analyzer report errors with the help of other phases.

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk Whatsapp# 0346-5100010



Tokens, Patterns and Lexemes

Following three are related but distinct terms in Lexical Analysis.

Token:

It is a pair consists of

- Token name
- Attribute value(optional)

Pattern:

- It is the description of the form that lexemes of a token may take.
- Pattern of a keyword is same as the keyword.
- For identifier and other tokens it can be a complex structure.
- For specifying lexeme patterns regular expressions are used.

Lexemes:

- Sequence of characters in the source program that matches the pattern for a token is called "lexeme" e.g.

Assignment statement A=B+C;

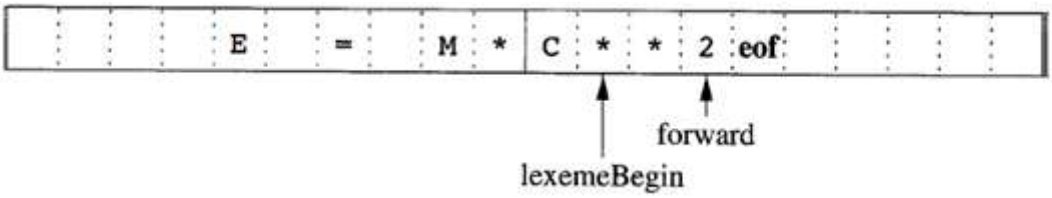
Lexemes	Token
A	<id , 1>
=	<=>
B	<id , 2>
+	<+>
C	<id , 3>

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk Whatsapp# 0346-5100010

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	< or > or <= or >= or == or !=	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

Input Buffering:

- Buffer is of the size N and N is usually the size of a disk block e.g 4096.
- Read N characters into a buffer.
- If file has less than N characters than a special characters "eof" is stored at the end of file.
- Two pointers to the input are maintained
  - **i)** Pointer lexeme begin marks the beginning of current lexeme.
  - **ii)** Pointer forward scans ahead until a pattern is found.
- Once the next lexeme is determined
  - **i)** Forward is set to the character at its right end.
  - **ii)** Lexeme begin is set to the character immediately after the lexeme just found.
- If we reached the end of one buffer then reload the other buffer with input.



Recognition of Tokens:

- Regular expressions are important notation for specifying lexeme patterns.
- Regular expressions effectively specify types of patterns that we actually need for tokens.
- They cannot express all possible patterns.

Regular Expression Of C Identifiers:

(letter| _)(letter | _ |digit)*

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk Whatsapp# 0346-5100010

Regular Definition:

- We give name to certain regular expression and use those names in subsequent expressions.
- Now we can represent above RE as follows

letter → A|B|…|Z|a|b|…|z|
digit → 0|1|…|9
id →(letter | _)(letter | _ | digit)*
It can also be written as:
letter→ [A-Z a-z-]
digit → [0-9]
id→ letter-(letter |digit)*

Regular Definition for Unsigned Number is

digit→ 0|1|….|9
digits→ digit digit *
optionalFraction→ .digits |E
optionalExponent→ (E (+|-|Ɛ) digits) |Ɛ
number → digits optionalFraction optionalExponent

It can also be written as:

digit→ [0-9]
digits→ digit +
number→ digits(.digits)? (E[+]? digits)?

Lexical Analyzer stripping out white space by recognizing the "token" white space defined by
ws→ (blank|tab|newline)+

Patterns for Tokens:

digit → [0-9]
digits → digit+
number→ digits(.digits)? (E[+-]?digits)?
letter- →[A-Za-z]
id →(letter|_)(letter | _ |digit)*
if →if
then→ then
else→else
relop→<|>|<=|>=|= = |!=

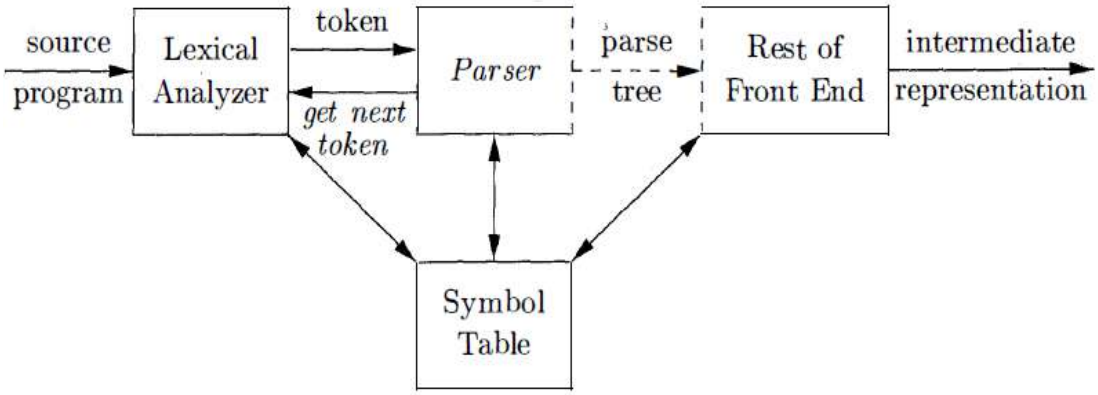
Grammar for branching statements

stmt→ if expr then stmt
| if expr then stmt else stmt
| Ɛ
expr→ term relop term | term
term→ id |number

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk Whatsapp# 0346-5100010

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
if	if	–
then	then	–
else	else	–
Any *id*	id	Pointer to table entry
Any *number*	number	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Role of Syntax Analyzer



- This phase is also called parser.
- It obtains tokens from LA then checks structure of tokens and produces a parse tree.
- Uses grammars to check structure/syntax.
- Recognizes correct syntax and find syntactic errors.

Syntax Error Handling:

Lexical: => Misspelling of identifiers.
=> Missing around text string.
Syntactic: => Misplaced.
=> Extra or missing braces.
=> Use of "case" without "switch".
=> Wrong order of tokens.
Semantic: => Type mismatch between operators and operands.
=>Use "return" in avoids type function.
Logical: => Errors in the logic of program.

Syntax Definition

- **CFGs** are used to define syntax of a language.

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk Whatsapp# 0346-5100010

- A grammar naturally defines the hierarchical structure of most programming language constructs.