Compiler design

Language Processors:

Compiler:

- A compiler is a program that can read a program from the source language and translate it into target language.
- A compiler can also report any error in the program during the translation.

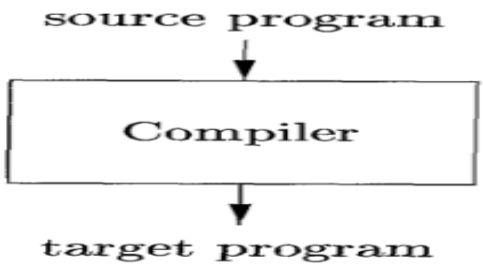
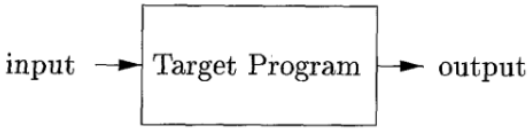


Figure 1.1: A compiler

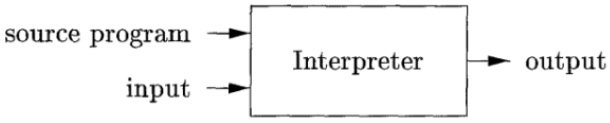
Target program:

target program is a program in which the user can call in order to process the input and produce the output

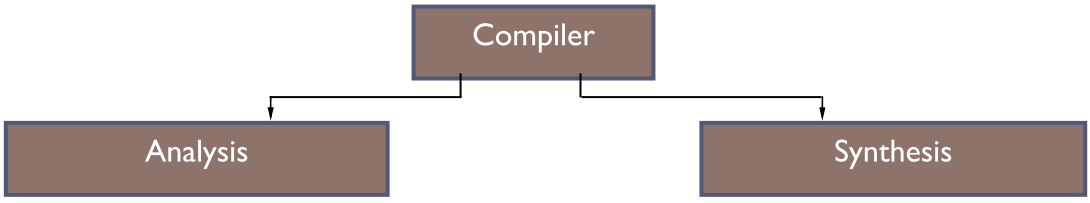


Interpreter:

An interpreter, like a compiler, translates high-level language into low-level machine language.



Structure of a Compiler consist of two phases *:*



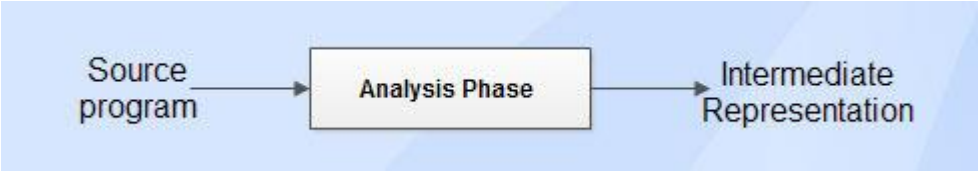
Analysis of the source program
Synthesis of a machine-language program

- **The analysis phase**:

  . it breaks the source program and creates an intermediate representation of the given source code.

  • It is also termed as front end of compiler.

  • Information about the source program is collected and stored in a data structure called symbol table.



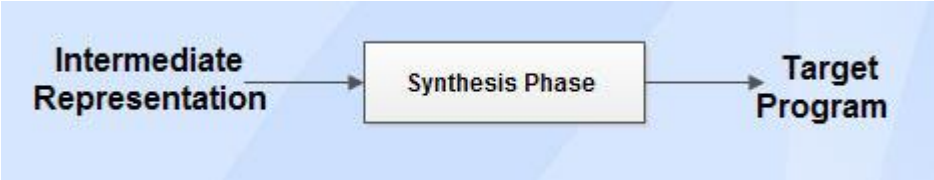
- **The synthesis phase**

➢ creates an equivalent target program from the intermediate representation.

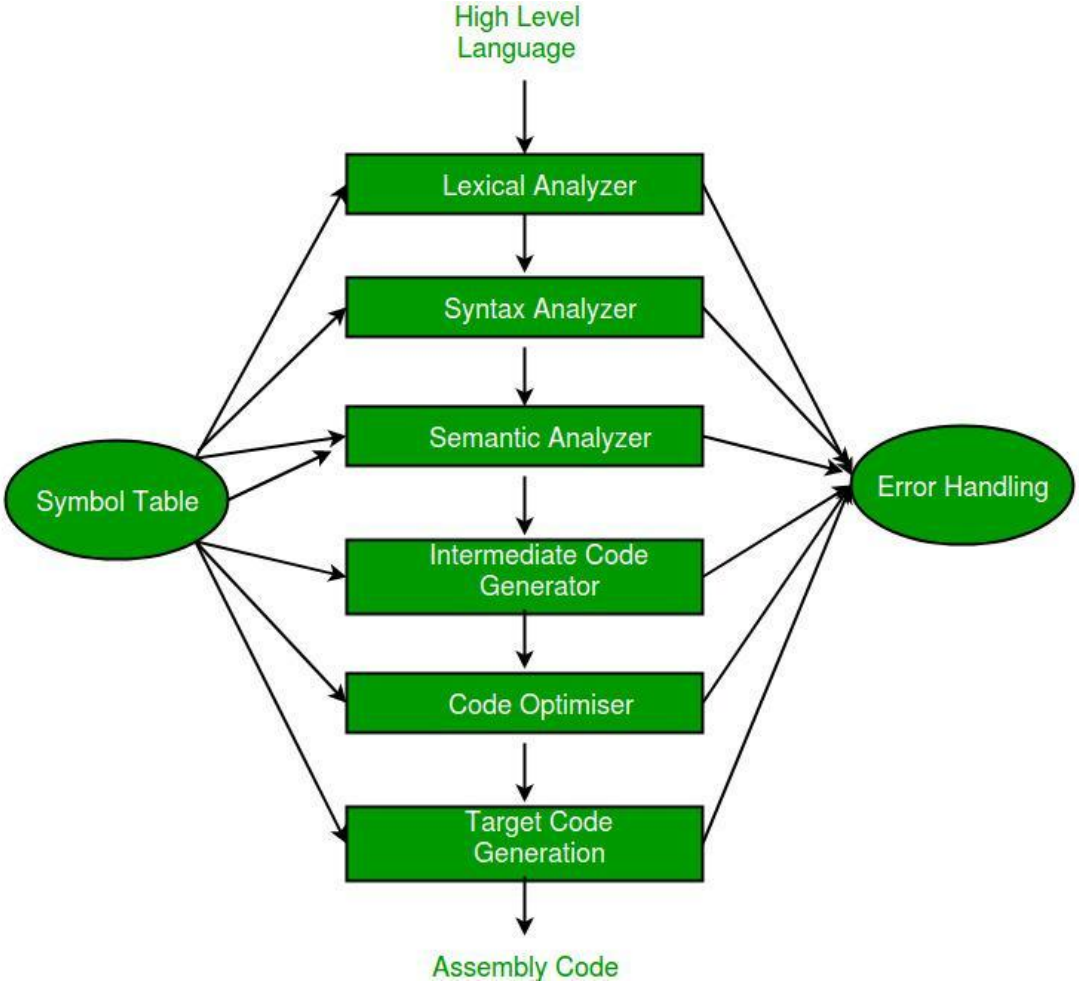
➢ Synthesis part takes the intermediate representation as input and transforms it to the target program.

➢ It is also termed as back end of compiler.

rrd



■ The design of compiler can be decomposed into several phases, each of which converts one form of source program into another.



The phases of compiler are as follows:

- Lexical analyzer
- semantic analyzer
- syntax analyzer
- intermediate code generator.
- Code optimiser
- Target code generation

Lexical Analysis:

Lexical analyzer phase is the first phase of the compilation process. It takes source code as input. It reads the source program one character at

a time and converts it into lexemes. Lexical analyzer represents these lexemes in the form of tokens.

Syntax Analysis:

Syntax analysis is the second phase of the compilation process. It takes tokens as input and generates a parse tree as output. In the syntax analysis phase, the parser checks that the expression made by the tokens is correct or not.

Semantic Analysis:

Semantic analysis is the third phase of the compilation process. It checks whether the parse tree follows the rules of language. Semantic analyzer keeps track of identifiers, their types and expressions. The output of the semantic analysis phase is the annotated tree syntax.

Intermediate Code Generation:

In the intermediate code generation, the compiler generates the source code into the intermediate code. Intermediate code is generated between the high-level language and the machine language. The intermediate code should be generated in such a way that you can easily translate it into the target machine code.

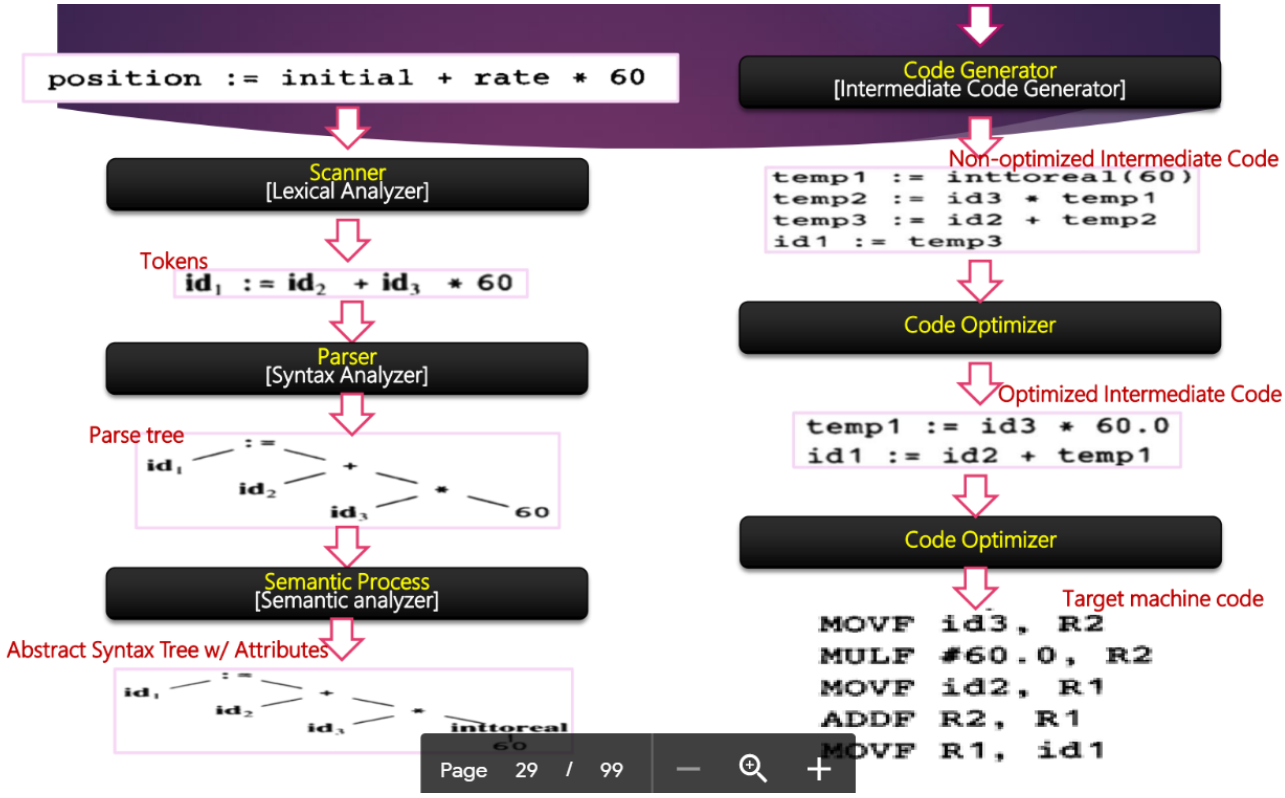
Code Optimization:

Code optimization is an optional phase. It is used to improve the intermediate code so that the output of the program could run faster and take less space. It removes the unnecessary lines of the code and arranges the sequence of statements in order to speed up the program execution.

Code Generation:

Code generation is the final stage of the compilation process. It takes the optimized intermediate code as input and maps it to the target machine language. Code generator translates the intermediate code into the machine code of the specified computer.

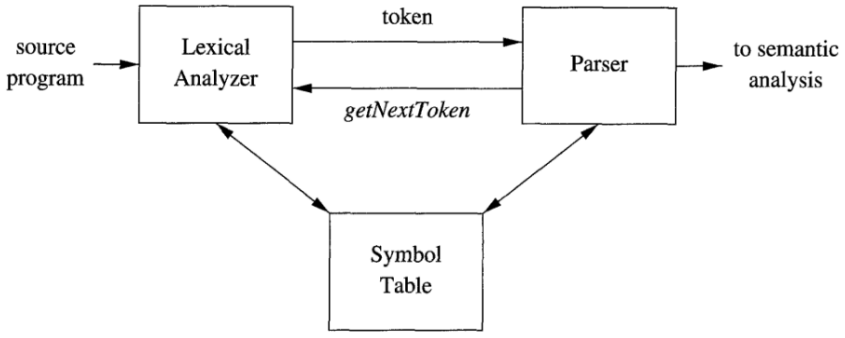
Example:


position := initial + rate * 60
Scanner
[Lexical Analyzer]
Tokens
id1 := id2 + id3 * 60
Parser
[Syntax Analyzer]
Parse tree
Semantic Process
[Semantic analyzer]
Abstract Syntax Tree w/ Attributes
Code Generator
[Intermediate Code Generator]
Non-optimized Intermediate Code
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
Code Optimizer
Optimized Intermediate Code
temp1 := id3 * 60.0
id1 := id2 + temp1
Code Optimizer
Target machine code
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
Page 29 / 99

.....

Q)Explain about the role of lexical analyzer. Why the analysis phase of compiler is separated into lexical analysis and parsing?

The Role of the Lexical Analyzer:



The role of Lexical Analyzer in compiler design is to read character streams from the source code,check for legal tokens,and pass the data to the syntax analyzer when it demands.

-> As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program.
The stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well. When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.
These interactions are suggested in Fig. 3.1. Commonly, the interaction is implemented by having the parser call the lexical analyzer. The call, suggested by the getNextToken command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.

Lexical analysis separated from syntax analysis because of following reasons:(4-5 points enough)

1)Simpler design.

- Separation allows the simplification of one or the other.

- Example: A parser with comments or white spaces is more complex.

2)Compiler efficiency is improved.

- Optimization of lexical analysis because a large amount of time is spent reading the source program and partitioning it into tokens.

3)Compiler portability is enhanced.

- Input alphabet peculiarities and other device-specific anomalies can be restricted to the lexical analyzer.

4)Because it makes both analyzers easier to debug.

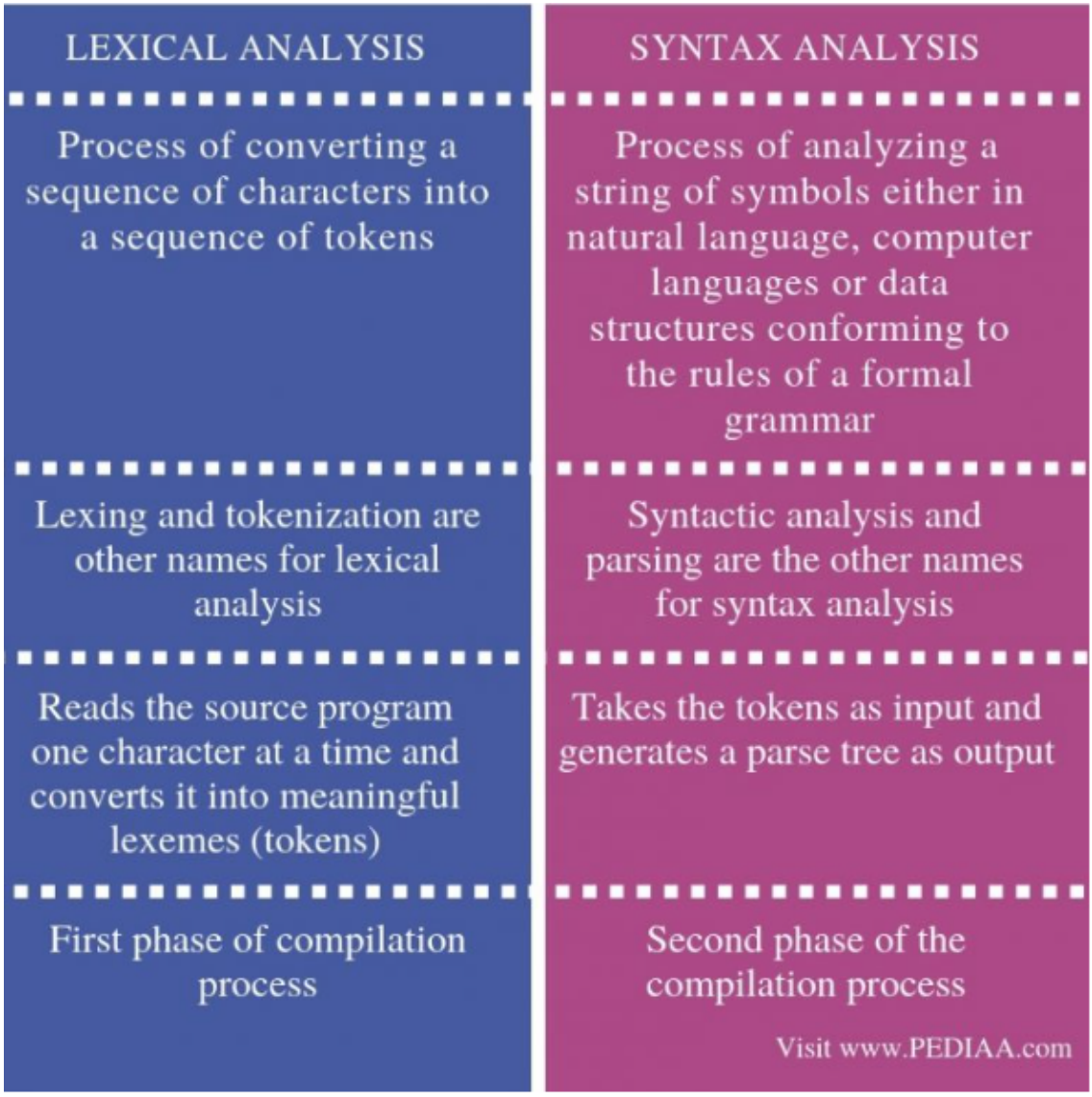
5)Lexical analysis is limited to just separating and identifying the token stream. It doesn't matter if the syntax is wrong. That is not the responsibility of lexical analysis. And even then, it is usually necessary to look at two (or more) tokens to determine if a syntax error has occurred.

6)Having the two separated means that You can add new syntax without ever changing the lexical analyser - for example add a new use for an existing keyword.

7)Having the two separated means that You can change the lexical definition of the language (for example change what you expect a valid identifier is) without breaking the syntax.

8)Having the two separate means that you can have the definition for each and have the lexical analyser and the syntax analyser generated from definitions.

Lexical Analysis vs Parsing (*syntax analysis*) :



Q)Define Lexeme and Token. Identify the lexemes and token from the program fragments given below
printf(“The value is %d”, value);

SOL)

Identifying the lexemes:

printf, (, "The value is %d" , value , comma ,)

Identifying the tokens:

<printf>
<(>
<literal,“The value is %d”>
<op,”,”>
<id,value>
<)>

Token: Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are,
1)Identifiers 2) keywords 3) operators 4) special symbols 5)constants

Pattern: A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Lexeme: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Token	lexeme	pattern
const	const	const
if	if	if
relation	<,<=,= ,< >,>=,>	< or <= or = or < > or >= or letter followed by letters & digit
i	pi	any numeric constant
nun	3.14	any character b/w “and “except"
literal	"core"	pattern

Q)Describe the strategies available for the *parser* to recovery from the detected error. link

Different Error recovery methods in Compiler Design are:

- **Panic mode recovery**
- **Statement mode recovery**
- **Error productions**
- **Global correction**

1)panic mode:

- **In this as soon as the parser detects an error anywhere in the program it immediately discard the rest of the remaining part of code.**
- **This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.**

2)Statement Mode recovery

- **In this method, when a parser encounters an error, it performs necessary correction on remaining input so that the rest of input statement allow the parser to parse ahead.**
- **The correction can be deletion of extra semicolons, replacing comma by semicolon or inserting missing semicolon.**
- **it prevents the parser from developing infinite loops.**

3)Error productions

If a user has some knowledge of errors that are very common then these type of errors are recovered by increasing the grammar with error production that produce incorrect constructs. It is very difficult and problematic to maintain.

4) Global correction

- **The parser examines the whole program and tries to find out the closest match for it which is error free.**
- **The closest match program has less number of insertions, deletions and changes of tokens to recover from erroneous input.**
- **Due to high time and space complexity, this method is not implemented practically.**

Q)Describe the strategies available for the *lexical analyzer* to recover from the detected error?

```
fi ( a == f(x)) ...
```

a lexical analyzer cannot tell whether `fi` is a misspelling of the keyword `if` or an undeclared function identifier. Since `fi` is a valid lexeme for the token **id**, the lexical analyzer must return the token **id** to the parser and let some other phase of the compiler — probably the parser in this case — handle an error due to transposition of the letters.

However, suppose a situation arises in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input. The simplest recovery strategy is "panic mode" recovery. We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left. This recovery technique may confuse the parser, but in an interactive computing environment it may be quite adequate.

Other possible error-recovery actions are:

1. Delete one character from the remaining input.
2. Insert a missing character into the remaining input.
3. Replace a character by another character.
4. Transpose two adjacent characters.

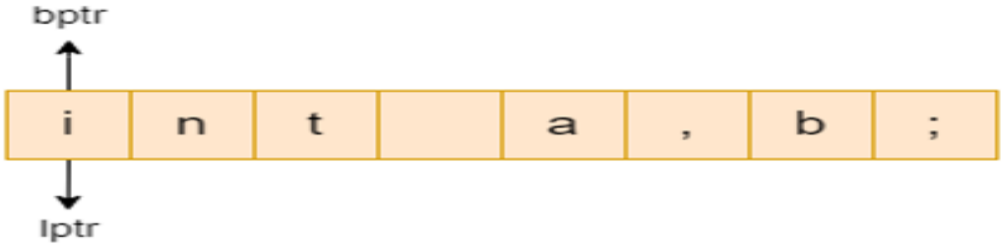
Transformations like these may be tried in an attempt to repair the input. The simplest such strategy is to see whether a prefix of the remaining input can be transformed into a valid lexeme by a single transformation. This strategy makes sense, since in practice most lexical errors involve a single character. A more general correction strategy is to find the smallest number of transformations needed to convert the source program into one that consists only of valid lexemes, but this approach is considered too expensive in practice to be worth the effort.

What is Input Buffering in Compiler Design (or) How buffer pair use lexical analyzer

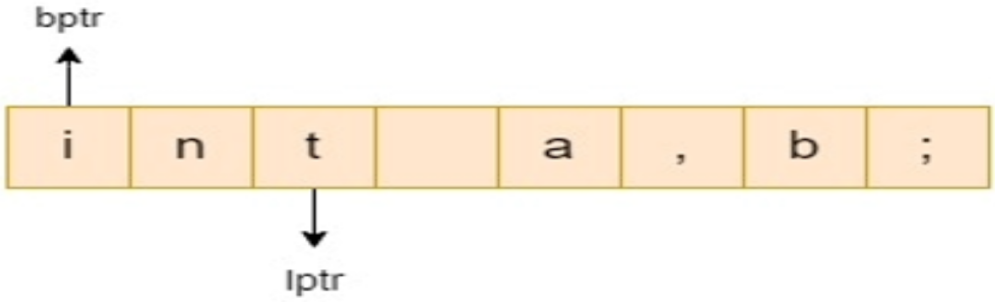
link

- The lexical analyzer scans the input from left to right one character at a time.
- It uses two pointers:.
  - The Begin Pointer (bptr) is a pointer that points to the start of the string to be read.
  - Look Ahead Pointer(lptr) continues its hunt for the token's end.

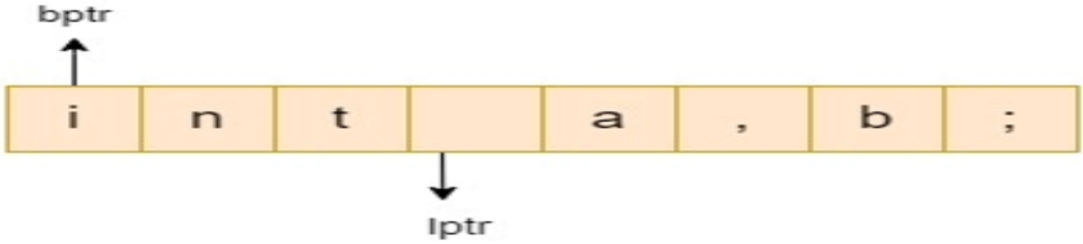
- **Example:** For the statement int a,b;



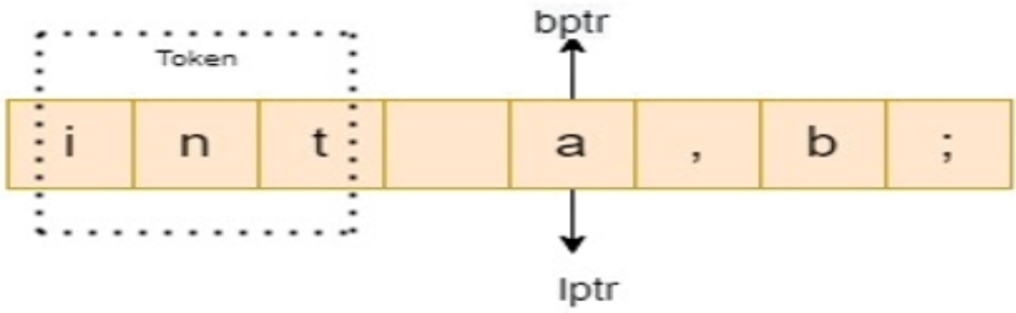
- Both points begin at the start of the string that is saved in the buffer.
- The Look Ahead Pointer examines the buffer until it finds the token.



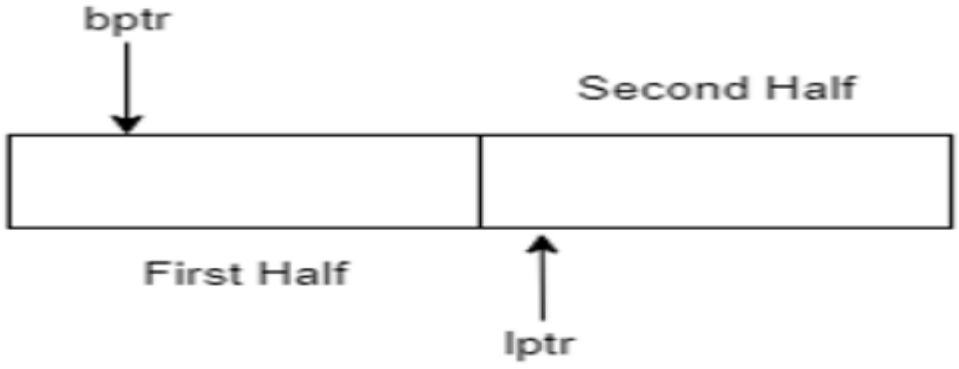
- Before the token ("int") can be identified, the character ("blank space")must be checked before the token ("int")



- Both pointers will be set to the next token ('a') after processing the token ("int"), and this procedure will be continued throughout the program.



- A buffer can be divided into two halves.
  - ➔ If you move the look Ahead cursor to the first half, the second half will be filled with fresh characters to read.
  - ➔ If you shift the look Ahead cursor to the second half's buffer, the first half will be filled with new characters, and so on.



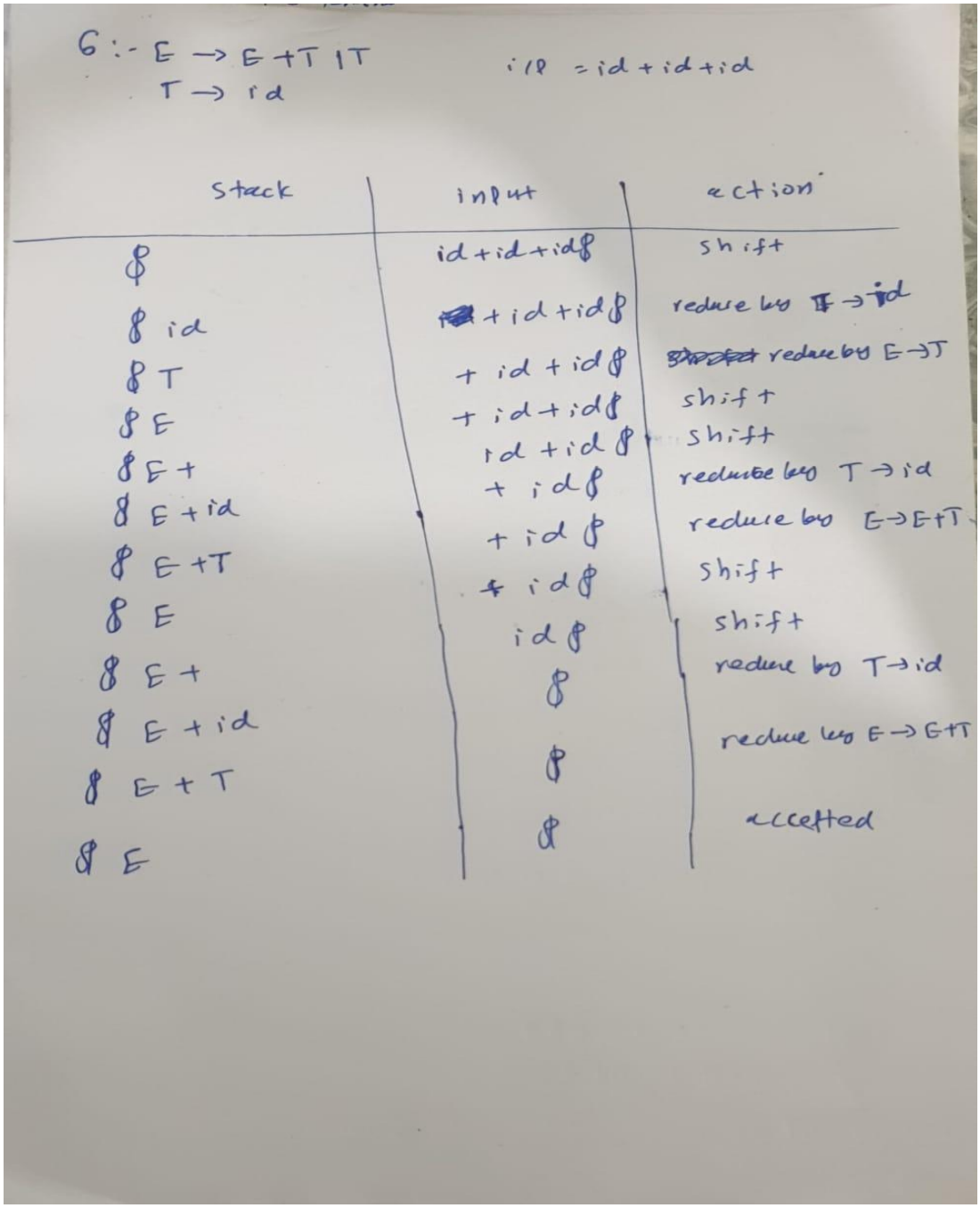
Input Buffering

……..

c) Show the moves made by the stack of a shift reduce parser for accepting the input " id+id+id"

G: E->E+T | T

T->id



Q)Eliminate the left recursion from the following grammar.

Online converter link
S->A | B
A->Aa | ε
B->Bb| Sc | ε

sol)
S->A | B no LR
A->A'
A'->aA'
B->ScB'|εB'
B'->bB'|ε

(other solution)
subtituting
S -> A | B
A -> Aa | e
B -> Bb | Ac | Bc | e

converting
S -> A | B
A -> A'
A' -> aA' | e
B -> AcB' | B'
B' -> bB' | cB' | e

……

Q)Construct the Transition Diagram for accepting the given operators.
+, -, *, /, ++, --

???

……
Q)Construct a LL(1) parsing table for the grammar given below. Check whether the grammar is LL(1) or not.
S-> +TS |)S | ε T-> (X) X->TX | [X] | y

……

Q)**Most languages are case sensitive, so keywords can be written only one way, and the regular expressions describing their lexeme is very simple. However, some languages, like SQL, are case insensitive, so a keyword can be written either in lowercase or in uppercase, or in any mixture of cases. Thus, the SQL keyword SELECT or FROM can also be written select, FrOm, From, Select, or sElEcT, for instance.**
Write a regular expression for the SQL command in a case insensitive language. Illustrate the idea by writing the regular expression for "select attribute_name from
table_name" in SQL.

soL)

II. **Most languages are case sensitive, so keywords can be written only one way, and the regular expressions describing their lexeme is very simple. However, some languages, like SQL, are case insensitive, so a keyword can be written either in lowercase or in uppercase, or in any mixture of cases. Thus, the SQL keyword SELECT can also be written select, Select, or sElEcT, for instance. Show how to write a regular expression for a keyword in a case insensitive language. Illustrate the idea by writing the expression for "select" in SQL.**

Answer

```
select -> [Ss][Ee][Ll][Ee][Cc][Tt]

OR

Select→(S|s)(E|e)(L|l)(E|e)(C|c)(T|t)
```

unit1:
Explain how the front end of a compiler works by translating the given statement Area=length*breadth;

Describe why left factoring is required for implementing top down parser?

Define the following with proper examples:
i) Regular expressions ii) Languages.
Show ambiguity in grammar with an example grammar and a proper input string.