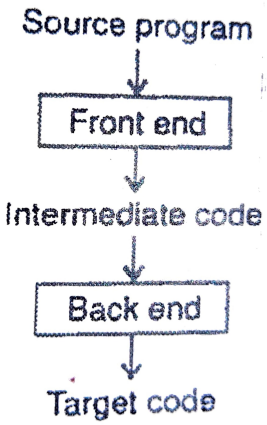
Introduction to Compiler Construction

- **Compiler**: Compilers are computer program that translates one language to another.
- **Two Aspects of Compilation**:
  - Generate code to implement semantic of source program in execution domain.
  - Provide diagnostics for violation of PL semantics in source program.
- **Interpreter**: It translates one statement of High Level Language into machine code, then executes the resulting machine code, then translates the next instruction of HLL, executes it, and son on. It never translates the whole program of HLL at one time.

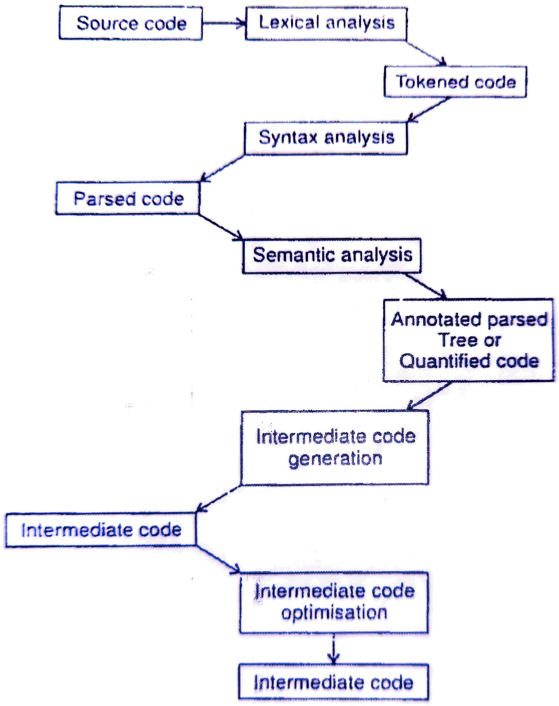
Compiler	Interpreter
Translate high level instructions into machine language.	Translate HLL into intermediate code and executes the program.
The program in whole is compiled at a time and then only be executed, once translation is complete.	Statements from source program are fetched and executed one by one.
Once a compiled program is exists, we can re -run it any time we want to , without having to use the compiler each time.	No copy of translation exists, and if the program is to be re-run, it has to be interpreted all over again.
If any changes done to the source code, it requires a complete recompilation.	It is often quicker and easier to make changes with a program interpreted then compiled one and son development time may be reduced.
Results in a faster running program.	Results in comparatively slow executing program.

- **Structure of Compiler**:



- Front end (Analysis) and Back end (synthesis)

Phases of Compiler (Analysis)

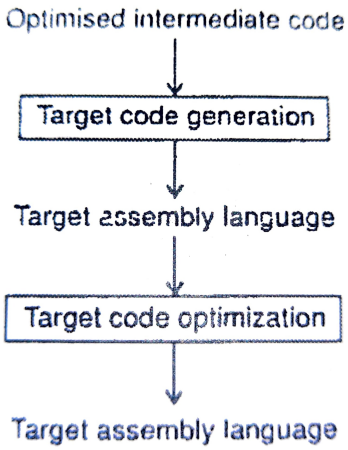


1. **Lexical Analysis**:
   - First phase of front end and so of compiler.
   - Lexical analyzer reads input as a character stream and breaks it up into smaller meaningful character sequence called **Tokens or lexemes**.
   - This phase is also known as **scanning.**
2. **Syntax Analysis**:

- This phases takes the list of tokens produced by the lexical analysis and arranges these in a tree structure that reflects the structure of the program.
- This phase is often called **parsing.**

3. **Semantic Analysis**:
   - This phase analyses the syntax tree to determine if the program violates certain consistency requirement.
   - An important part of semantic analysis is **type checking**, so sometime this phase is also called **type checking phase**.
4. **Intermediate Code Generation**:
   - By using annotated parse tree ,this phase generate intermediate code.
5. **Intermediate Code Optimization**:
   - The intermediate code is optimized by using various techniques, like elimination of common sub-expressions, elimination of unreachable code segments, elimination of statements that remain unmodified in the loop etc.

- **Phases of Compiler (Synthesis)**



  - **Target Code Generation**: In this phase, the intermediate code is translated into machine or assembly code.
  - **Target Code Optimization**: This phase transforms target code into more efficient, optimized target code.
- **Error Handling**: While compilation compiler detects compiler time as well as run time error and generates appropriate error messages.

- **One Pass Compiler**: While translating, source program into target program, compiler entire program in one single pass.

  One pass compilers are faster but produce inefficient target code.

- **Multipass Compiler**: While translating source program into target program, compiler reads entire program multiple time for each pass.

  Multipass compiler takes longer time to process but produces efficient target code.

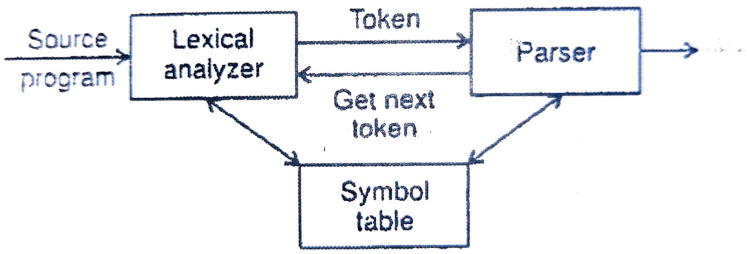
- **Cross Compiler**: A complier that generate code for a different machine, from the one on which it runs called cross compiler.

- **Bootstrapping**: It is a process in which we write a compiler in language A (let it target B) and then let it compile itself to produce a compiler from A to B written in B.

  Bootstrapping is useful in implementation of cross compiler.

Lexical Analysis

- **Output of Lexical Analyzer**: It reads input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.



- **Token**: The lexical analyzer reads a source program one character at a time and translates it into a sequence of primitive units called tokens.
- **Pattern**: The set of strings in the input for which the same token is produced as output. The set of strings is described by a rule called pattern.
- **Lexeme**: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.
- **Install_id()**: return attribute value.
- **gettoken()**: Used to obtain the token.
- **atoi**: Converts string to integer.
- **Input Buffering:**
  - Here a two buffer input scheme is useful when look ahead on the input is necessary to identify token.
  - To speed up the lexical analyzer sentinels are used to mark the buffer end.
  - Three approaches to implement a lexical analyzer.
    1. Use of lexical analyzer generator.
    2. Write lexical analyzer in conventional programming language, using the I/O facilities of that language to read the input.
    3. Write it in assembly language and explicitly manage the reading of input.
- **Three Sections of Lex Program:**

declarations

%%

translation rules

%%

auxiliary procedures

1. **Declarations Section**: It includes declarations of variables, manifest constant and regular definition.
2. **Translation Rules Section**: The translation rules of LEX program takes the form

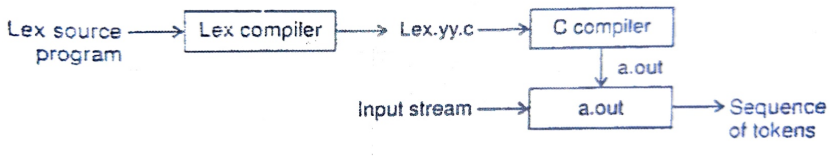
   P1 {action1}

   .

   Pn {action n}

   Each Pi is a regular expression and each action i is a program fragment.
3. **Auxiliary Procedures Section**: This section holds the different procedures needed by the actions. Precompiled procedures can also be included in it.

- LEX compiler gives Lex.yy.c output.
- **LEX Library Function:**
  1. **yylex()** : It scans number of lines from the keyboard.
  2. **yywrap()** : If the value returned from this function is 1 then it indicates that no further input is avaliabe.
  3. **yyerror()** : Indicates the error in lex programs.
- **Execution of LEX Program:**



💡 Solve LEX program questions from 2-27 onwards

Syntax Analysis

- **Syntax Analysis** is the second stage in compilation of a source program after lexical analysis. Syntax analysis verifies if the tokens produced by lexical analyzer are properly sequenced in accordance with the grammar of the language.

Top Down Parser	Bottom Up Parser
It uses derivation process.	It uses reduction process.
It has two types: Backtracking Parsing and Predictive Parsing.	It has two types: Operator Precedence Parsing and LR Parsing.
Predictive Parsers scans Left to Right Left Most derivation(LL).	Predictive Parsers scans Left to Right Right Most derivation(LR).
It does not accept ambiguous grammars.	It accepts and deals with ambiguous grammars.
Backtracking can be avoided using left factoring grammar.	This parser cannot avoid backtracking.
No proper error indication is possible.	Errors are indicated properly.
It is slower than bottom-up.	It is faster than top-down.

💡 **Solve Top-Down Parser Questions**

1. Elimination of Left Recursion
2. Left Factoring
3. Recursive Descent Parser
4. First and Follow
5. Predictive Parser

Recursive Descent Parser	Predictive Parser
Does not use parse table to store grammar.	Uses parse table to store grammar.
Grammar is directly converted into program.	Grammar is directly stored into parse table.
Requires more space in memory.	Requires less space in memory.

Recursive Descent Parser	Predictive Parser
Recursive function are used in the parser.	Recursive functions are not used in the statement.
Requires more time to scan the input statement.	Requires less time to scan the input statement.
It does not use functions like FIRST() and FOLLOW().	It uses functions FIRST() and FOLLOW().

💡 **Solve Bottom-Up Parser Questions**

1. Leading and Trailing
2. Precedence Relation Table
3. LR(SLR) Parser
4. CLR Parser
5. LALR Parser

YACC

- **YACC** stands for Yet Another Compiler Compiler.
- YACC Library Functions:
  - **yyparse()** : Used to parse the input.
  - **yyerror()** : Called when an error is encountered.