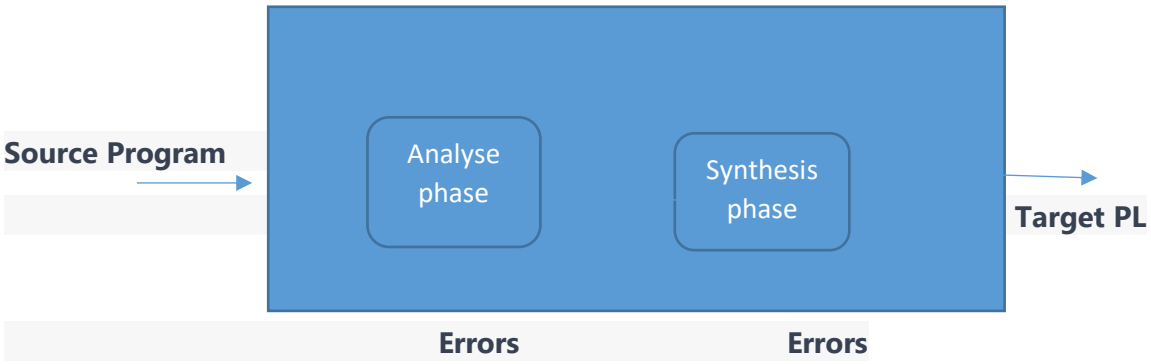
Que- explain the Fundamentals of language processing activities

Ans: The fundamentals of language processing are known as combination of Analysis of sourse program And syntheses of target program.

Language processing activities in system programming involve the manipulation and management of programming languages to create and maintain software systems.

These activities are essential for translating human-readable code into machine-executable instructions.



❖ **Analysis Consist of Three Rule**

Lexical Analysis (Scanning):

- Lexical analysis, also known as scanning or tokenization, is the first step in language processing.
- It involves breaking the source code into a sequence of tokens, which are the smallest meaningful units of a programming language (e.g., keywords, identifiers, operators, literals).
- Lexical analyzers generate a token stream as output.

Syntax Analysis (Parsing):

- Syntax analysis is the process of determining the grammatical structure of the source code.
- It uses context-free grammars to define the valid syntax of a programming language.
- A parser checks whether the token stream adheres to the language's grammar rules and generates a parse tree or an abstract syntax tree (AST) as a hierarchical representation of the code's structure.

Semantic Analysis:

- Semantic analysis checks the meaning and correctness of the code beyond its syntax.

- It verifies that the code adheres to the language's semantic rules, such as type checking and scoping rules.
- Semantic analyzers may also perform optimizations and generate intermediate code representations.

❖ **Syntheses of target program consist of two main activities**

Code Generation:

- Code generation translates the intermediate representation or the parse tree into machine code or assembly language for the target architecture.
- This step involves selecting appropriate machine instructions and managing memory allocation.
- Code generators need to take into account the specific features and constraints of the target hardware.

Intermediate Code Generation:

- Some compilers and language processors generate an intermediate representation of the code before translating it into machine code.
- Intermediate code is often easier to optimize and target multiple platforms.
- Examples of intermediate representations include three-address code, bytecode, or abstract syntax trees.

Properties

1-IR should be easy to construct and analysis

2-The IR should have processing effiencency

3-IR must be compact i.e memory efficiency.

Que- Explain Language Processing Activities

Answer:I) Language processing activity bridges the ideas of software designers with actual execution on the computer system.

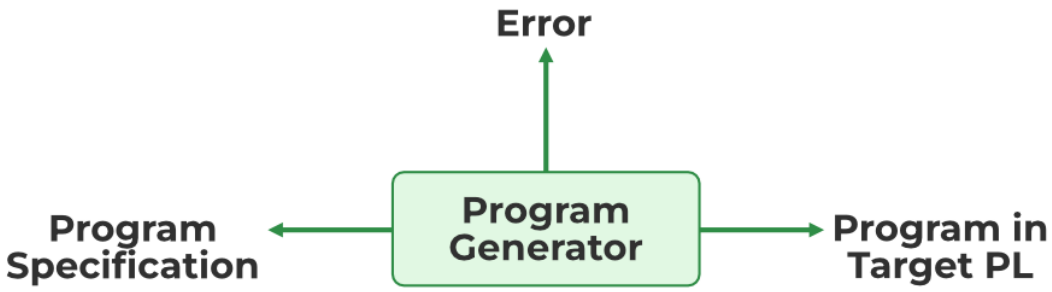
II) The designer expresses the idea related to the application domain of the software. To implement these ideas, the description of the ideas has to be interpreted as related to the execution domain of the computer system.

III) The Language Processing Activities divided into two parts

1-Program generation activities

2-Program execution activities.

1-Program generation Activities:



This activity generates a program from its specification program. Program generation activity bridges the specification gap.

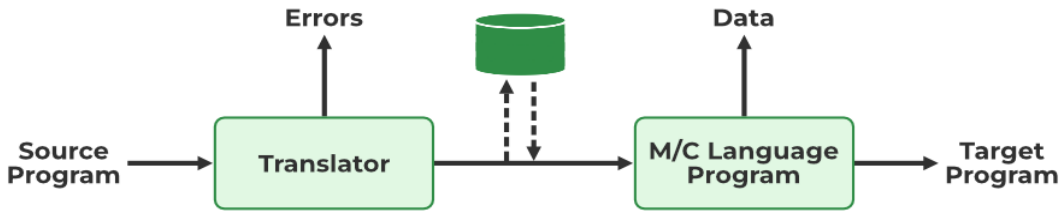
- A program generator is a software system program that accepts the specifications of the program to be generated and generates the program in the target program in a target programming language.
- This activity generates a program from its specification.
- The program generator introduces a new domain between the application and the programming language domain called the program generator domain.
- The generator domain is close to the application domain .It is easy for the designer or programmer to write the specification of the program to be generated.
- A Program generator is software that enables an individual to create a program with less efforts and prog.knowlege.
- A user may only be required to specify the step or rules required for the program and not need to write and code or less code.

2-Program execution activities :

1. This activity involves the actual execution of the program generated by the program generator. The program execution activity bridges the implementation gap.
2. The program execution activity involves loading the program into the memory of the computer system, interpreting the program instructions, and executing them on the computer system.
3. During program execution, the computer system reads the program instructions from memory and executes them one by one, performing the necessary computations and producing the desired results.
4. The program execution activity involves various components of the computer system, including the processor, memory, input/output devices, and other system resources required to execute the program.

Two popular models for program execution are- i) Program Translation
ii)Program Interpretation.

i)Program Translation:



- The Program translation model bridges the execution gap by translating a program written in a PL called source program into an eguivalent program in the machine language called target program(TP)
- **Characteristics**
  - A program must be translated before it can be executed
  - A translated program may save in a file. So that, we can execute saved files or programs repeatedly.
  - A program must be retranslate
- **Program Interpretation:**

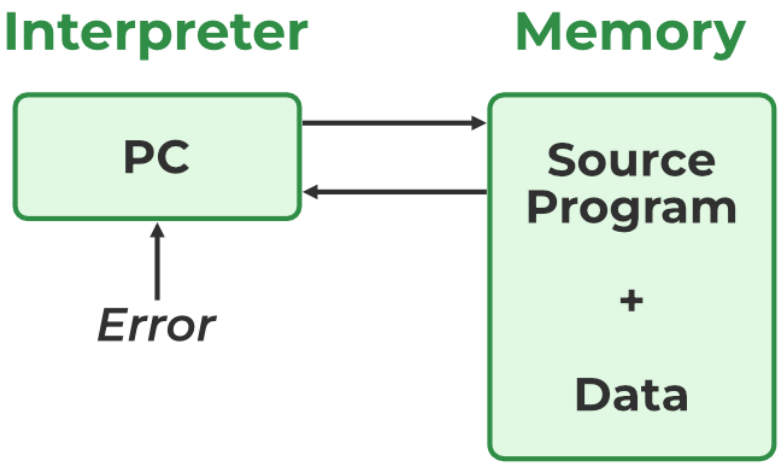
The interpreter reads the source program and stores it in its memory.

Throughout interpretation takes a statement, finds its meaning, and performs actions that implement this. This includes computational and input-output actions.

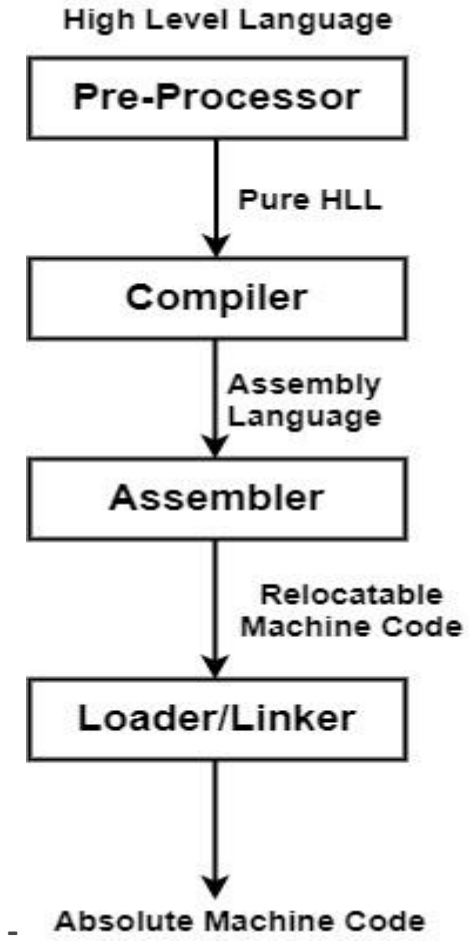
The CPU uses a program counter (PC) to note the address of the next instruction to be executed.

This Instruction is subjected to the instruction execution cycle consisting of the following steps:

- Fetch the instruction.
- Decode the instruction to determine the operation to be performed, and also its operands.
- Execute the Instruction



Que - What is Language Processor Systems ?



In a language processing system, the source code is first preprocessed.

The modified source program is processed by the compiler to form the target assembly program which is then translated by the assembler to create relocatable object codes that are processed by linker and loader to create the target program.

It is based on the input the translator takes and the output it produces, and a language translator can be defined as any of the Above diagram.

High-Level Language − If a program includes #define or #include directives, including #include or #define, it is known as HLL.

Pre-Processor − The pre-processor terminates all the #include directives by containing the files named file inclusion and all the #define directives using macro expansion. A pre-processor can implement the following functions −

- **Macro processing** − A preprocessor can enable a user to define macros that are shorthands for higher constructs.
- **File inclusion** − A preprocessor can include header files into the program text.
- **Rational preprocessor** − These preprocessors augment earlier languages with additional current flow-of-control and data structuring facilities.
- **Language Extensions** − These preprocessors try to insert capabilities to the language by specific amounts to construct in macro.

Pure HLL − It means that the program will not contain any # tags. These # tags are also known as preprocessor directives.

1. Compiler :

i)-The language processor that reads the complete source program written in high-level language as a whole in one go and translates it into an equivalent program in machine language is called a Compiler. Example: C, C++, C#.

ii)-In a compiler, the source code is translated to object code successfully if it is free of errors. The compiler specifies the errors at the end of the compilation with line numbers when there are any errors in the source code. The errors must be removed before the compiler can successfully recompile the source code again the object program can be executed number of times without translating it again.

Assembler − Assembler is a program that takes as input an assembly language program and changes it into its similar machine language code.

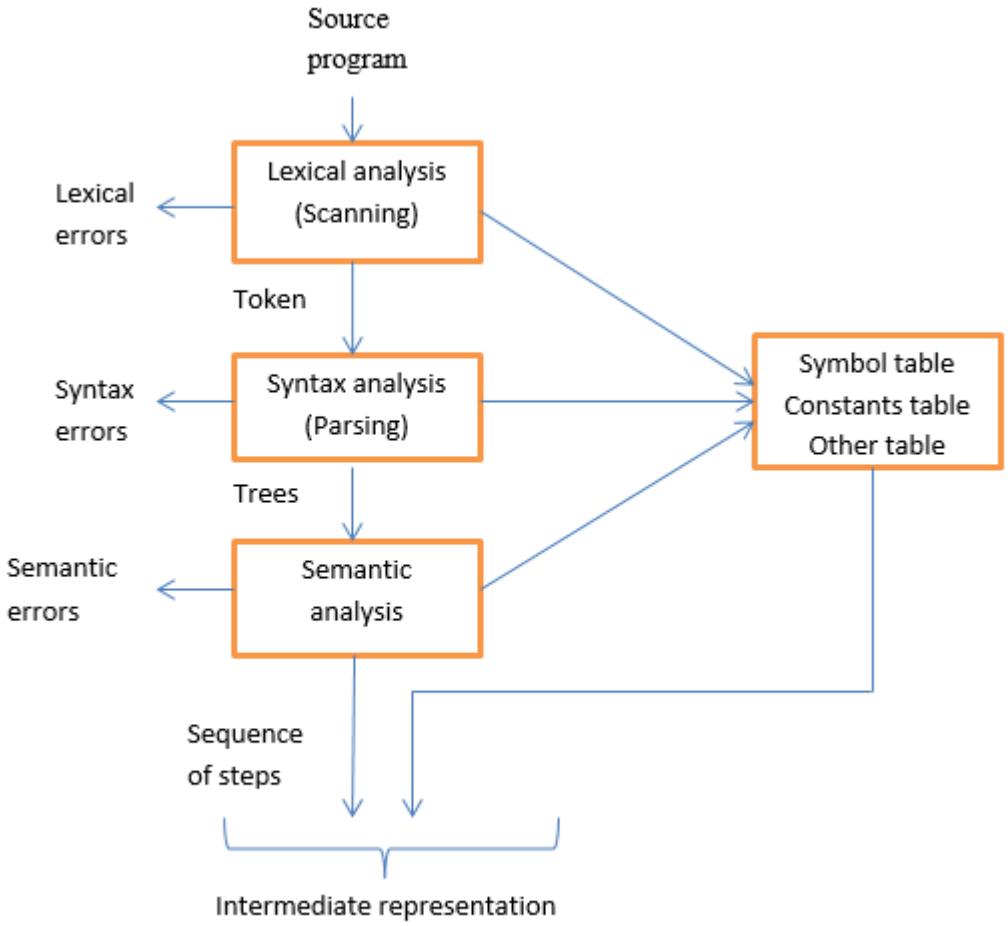
Assembly Language − It is an intermediate state that is a sequence of machine instructions and some other beneficial record needed for implementation. It neither in the form of 0's and 1's.

Advantages of Assembly Language

- Reading is easier.
- Addresses are symbolic & programmers need not worry about addresses.
- It is mnemonics. An example we use ST instead of 01010000 for store instruction in assembly language.
- It is easy to find and correct errors.
- **Relocatable Machine Code** − It means that you can load that machine code at any point in the computer and it can run. The address inside the code will be so that it will maintain the code movement
- **Loader / Linker** – This is a code that takes as input a relocatable program and compiles the library functions, relocatable object records, and creates its similar absolute machine program.
- The loading includes taking the relocatable machine program, changing the relocatable addresses, and locating the modified instructions and information in memory at the suitable area
- Linking enables us to create a single program from several documents of a relocatable machine program. These documents can have been resulting in different compilations, one or more can be library routines supported by the system available to any code that requires them.

- In system programming, multiple source files or modules may be involved.
- Linking combines these separate modules into a single executable file or library.
- Loading is the process of placing the executable code into memory for execution.
- Linkers and loaders handle the resolution of external symbols and address relocation.

Que- Explain Toy Compiler



Front end of toy compiler

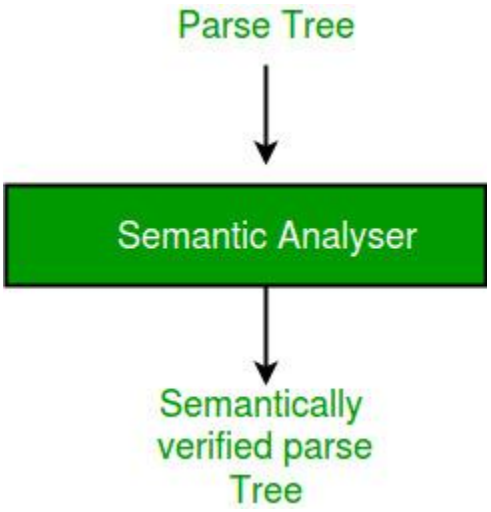
The compiler has two modules namely the front end and the back end. Front-end constitutes the Lexical analyzer, semantic analyzer, syntax analyzer, and intermediate code generator. And the rest are assembled to form the back end.

1. **Lexical Analyzer –**
   It is also called a scanner. It takes the output of the preprocessor (which performs file inclusion and macro expansion) as the input which is in a pure high-level language.
2. It reads the characters from the source program and groups them into lexemes (sequence of characters that "go together").
3. Each lexeme corresponds to a token. Tokens are defined by regular expressions which are understood by the lexical analyzer. It also removes lexical errors (e.g., erroneous characters), comments, and white space.

2. **Syntax Analyzer –**
It is sometimes called a parser. It constructs the parse tree. It takes all the tokens one by one and uses Context-Free Grammar to construct the parse tree.

Why Grammar?
The rules of programming can be entirely represented in a few productions. Using these productions we can represent what the program actually is. The input has to be checked whether it is in the desired format or not.
The parse tree is also called the derivation tree. Parse trees are generally constructed to check for ambiguity in the given grammar. There are certain rules associated with the derivation tree.

1. Any identifier is an expression
2. Any number can be called an expression
3. Performing any operations in the given expression will always result in an expression. For example, the sum of two expressions is also an expression.
4. The parse tree can be compressed to form a syntax tree



- **Semantic Analyzer –** It verifies the parse tree, whether it's meaningful or not. It furthermore produces a verified parse tree. It also does type checking, Label checking, and Flow control checking.
- **Intermediate Code Generator –** It generates intermediate code, which is a form that can be readily executed by a machine We have many popular intermediate codes.

- Example – Three address codes etc. Intermediate code is converted to machine language using the last two phases which are platform dependent.
  Till intermediate code, it is the same for every compiler out there, but after that, it depends on the platform. To build a new compiler we don't need to build it from scratch. We can take the intermediate code from the already existing compiler and build the last two parts.

- **Code Optimizer –** It transforms the code so that it consumes fewer resources and produces more speed. The meaning of the code being transformed is not altered. Optimization can be categorized into two types: machine-dependent and machine-independent.

- **Target Code Generator –** The main purpose of the Target Code generator is to write a code that the machine can understand and also register allocation, instruction

selection, etc. The output is dependent on the type of assembler. This is the final stage of compilation. The optimized code is converted into relocatable machine code which then forms the input to the linker and loader.

- **The back end**

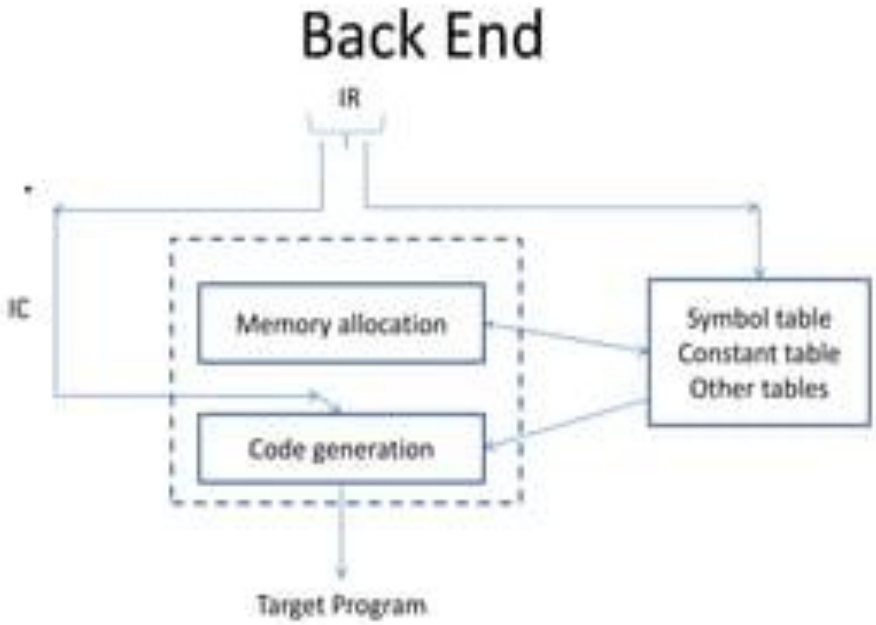


Fig: Back end of Toy Compiler

The backend perform two task as follows

- Memory Allocation
- Code Generation

Memory Allocation:

- Memory allocation is a simple task given the presence of the symbol table.
- The memory requirement of an identifier is computed from its type, length and dimensionality and memory is allocated to it.
- The address of the memory area is entered in the symbol table.

Code generation:

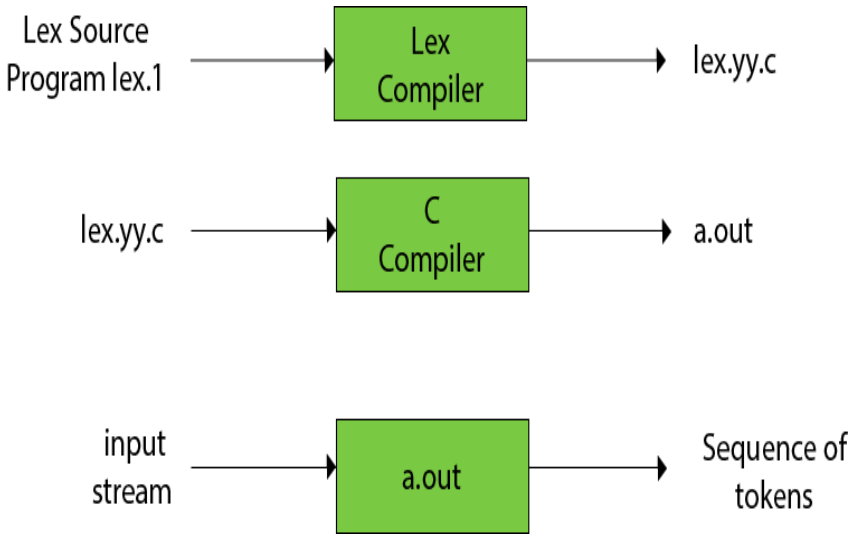
- Code generation uses knowledge of the target architecture.
- Knowledge of instruction and addressing modes in the target computer ,to select the appropriate instruction.
- The importatnt issues in code generation are:
- Determine the place where the intermediate results should be kept. either it is in memory location or in machine register.
- Determine which instruction should be used for type conversion operation.
- Determine which addressing modes should be used for accessing variables.

LEX Tool:

- Lex is a program that generates lexical analyzer. It is used with YACC parser generator.
- The lexical analyzer is a program that transforms an input stream into a sequence of tokens.
- It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

The function of Lex is as follows:

- Firstly lexical analyzer creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.
- Finally C compiler runs the lex.yy.c program and produces an object program a.out.
- a.out is lexical analyzer that transforms an input stream into a sequence of tokens.



Lex file format

A Lex program is separated into three sections by %% delimiters. The formal of Lex source is as follows:

```
    { definitions }
%%
     { rules }
%%
    { user subroutines }
```

Definitions include declarations of constant, variable and regular definitions.

Rules define the statement of form p1 {action1} p2 {action2}....pn {action}.

Where **pi** describes the regular expression and **action1** describes the actions what action the lexical analyzer should take when pattern pi matches a lexeme.

- Defines the rules program logic.

YACC TOOL

- YACC stands for **Yet Another Compiler Compiler**.
- YACC provides a tool to produce a parser for a given grammar.
- YACC is a program designed to compile a LALR (1) grammar.

- It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.
- The input of YACC is the rule or grammar and the output is a C program.

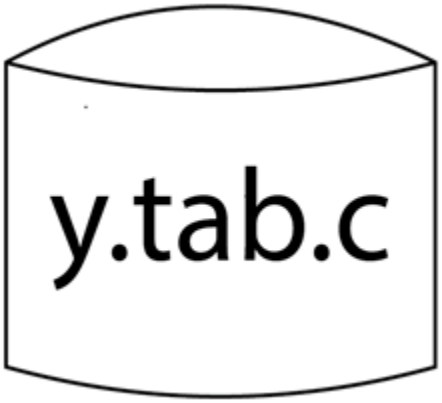
- These are some points about YACC
- **Input: A CFG- file.y**
- **Output: A parser y.tab.c (yacc)**
- The output file "file.output" contains the parsing tables.
- The file "file.tab.h" contains declarations.
- The parser called the yyparse ().
- Parser expects to use a function called yylex () to get tokens.

Basic Operational sequence:

1.This file contains the desired grammar in YACC format.

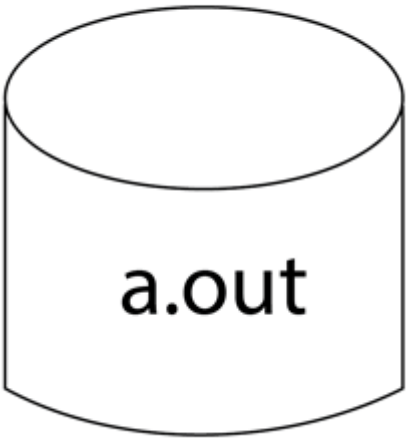
2. It shows the YACC program.

3. It is the c source program created by YACC



4. C Compiler

5. Executable file that will parse grammar given in gram.Y



Definations

Semantic Gap: The gap between application domain and execution domain called semnatic gap.

❖ Consequences of semantic gap

Large development time
Large effort
Poor quality of software

The specification gap is the symentic gap between application domain and execution domain.

Execution Gap:- It is the gap between symentic of program written in different programming languages.

Language processor: It Is a software which bridges specification or execution gap.

Language Translator:- Bridges execution gap between compiler and assembler

Source ⟶ Machine language.

Program Retranslator:- It bridges the same execution gap as translator but in reverse direction.

Language Migrator:- It bridges the specification gap between two programming languages.

Forward reference:- A forward reference of program entity is reference to entity which receives define in the program.