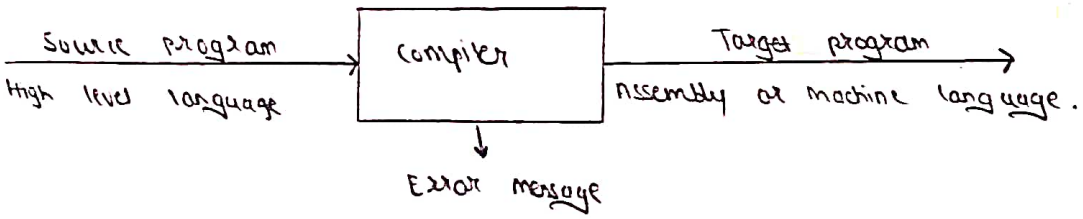
Practical No:1

Aim: To study phases of compiler.

Objectives:

- To study and understood the concept of compiler
- To study different phases of compiler



Date :

Practical No. 1

Aim: To study phases of compiler

OBJECTIVES :

- To Study & understand the concept of compiler.
- To study different phases of compiler.

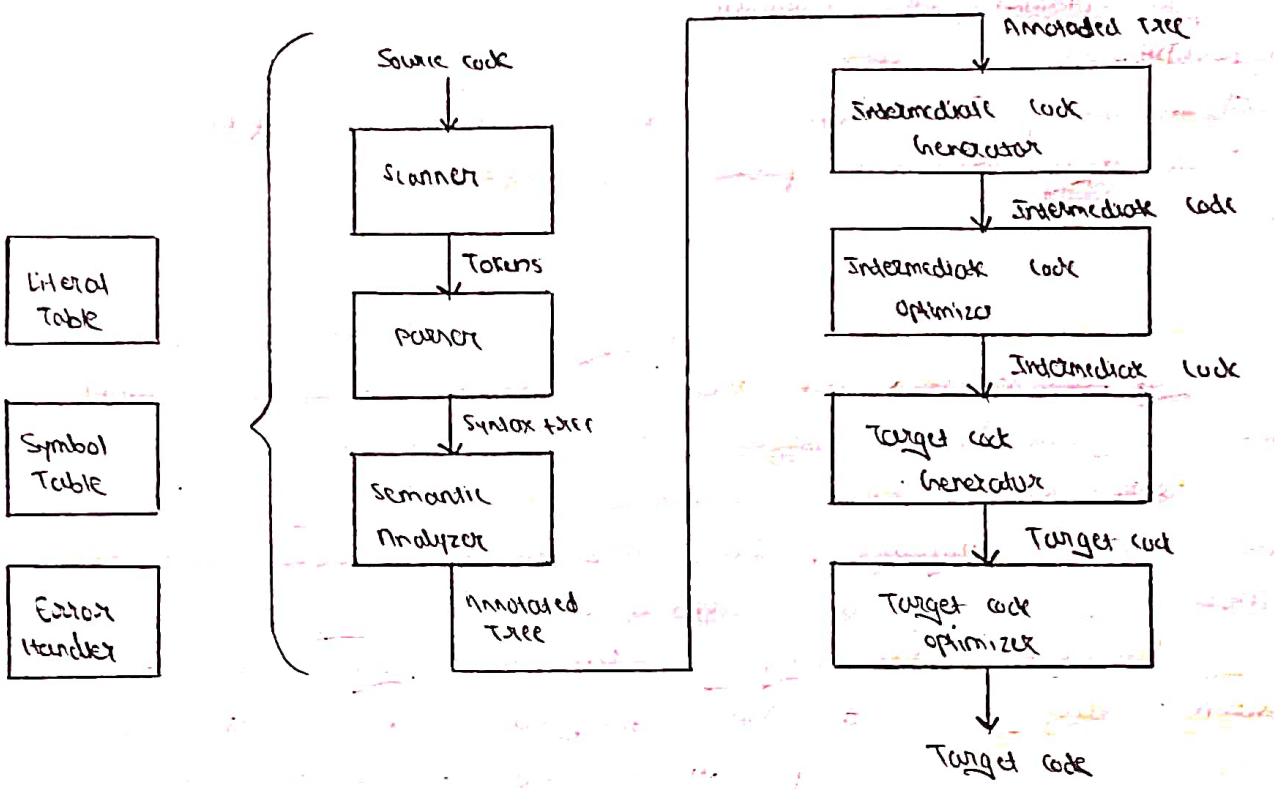
Theory :

In order to reduce the complexity of designing and building computers, nearly all of these are made to execute relatively simple commands. A program for a computer must be execute build by combining these very simple commands into a program in what is called machine language. Since this is a tedious and error-prone most programming is, instead, done using a high-level programming language. This language can be very different from the machine language that the computer can execute, so some means of bridging the gap is required. This is where the compiler comes in. A compiler translates a program written in a high-level programming languages that is suitable for human programmers into the low-level machine language that is required by computer. During this process, the compiler will also attempt to spot and report obvious programmer mistakes.

Using a high level language for programming has a large impact on how fast programs can be developed. The main reason for this are:

- Compared to machine language, the notation used by programming languages is closer to the way humans think about problems.
- The compiler can spot some obvious programming mistakes.
- Programs written in a high-level language tend to be shorter than equivalent programs written in machine languages.

Another advantages of using a high-level language is that the same program can be compiled to many different machine languages


Literal Table
Symbol Table
Error Handler
Source code
Scanner
Tokens
Parser
Syntax tree
Semantic Analyzer
Annotated tree
Annotated tree
Intermediate code generator
Intermediate code
Intermediate code optimizer
Intermediate code
Target code generator
Target code
Target code optimizer
Target code

Date :

and, hence, be brought to run on many different machines. On other hand, programs that are written in a high-level language and automatically translated to machine language may run somewhat slower than programs that are hand-coded in machine language. Hence, some time-critical programs are still written partly in machine language. A good compiler will, however, be able to get very close to the speed of hand-written machine code when translating well-structured programs.

The Translation Process:

A compiler performs two major tasks:

→ Analysis of the source program

→ Synthesis of the target-language instructions.

Phases of a Compiler:

Scanning
parsing
Semantic Analysis
Intermediate code Generation
Intermediate code optimizer
Target code Generation
Target code optimizer

Three auxiliary components interact with some or all phases:

literal Table
Symbol Table
Error Handler

a := x + y * 2.5 ;

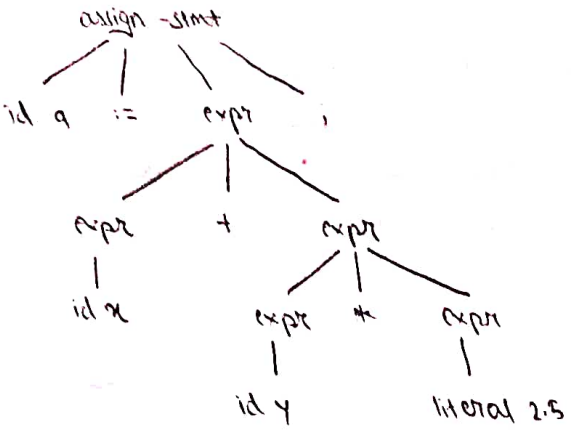


Fig. parse Tree

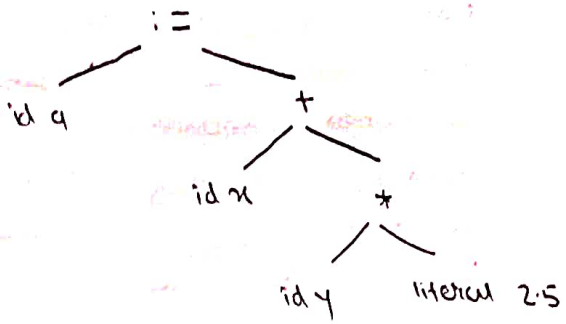


Fig. Syntax Tree

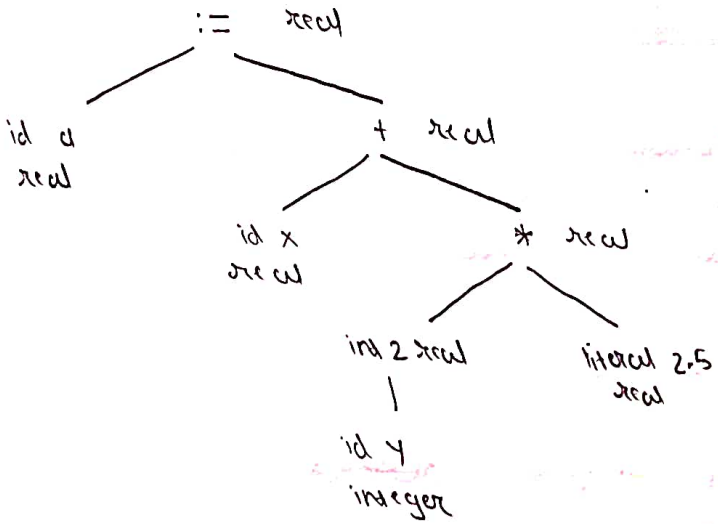


Fig: Annotated Syntax tree

Date :

Scanner :→

The scanner begins the analysis of the source program by:

- Reading file character by character.
- Grouping characters into tokens.
- Eliminating unneeded info. (comments and white space).
- Entering preliminary info. into literal or symbol table.
- processing compiler directives by setting flags.
- Tokens represent basic program entities such as:
  Identifier, literals, Reserved words, operators, delimiter, etc..

Example :

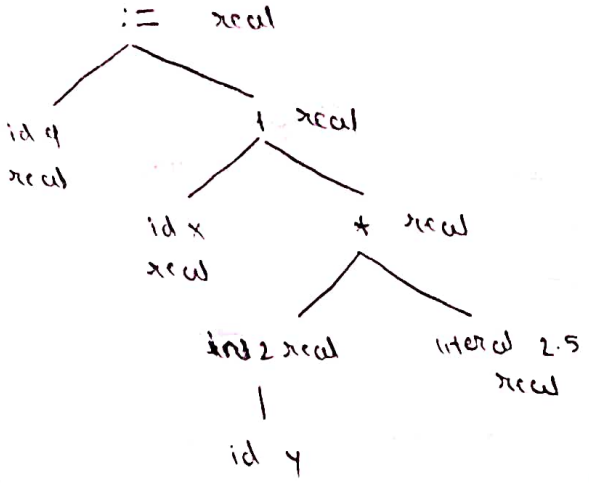
a := x + y * 2.5 ; is scanned as a identifier, := identifier
: = assignment operator, * multiplication operator, x
identifier, 2.5 real literal, + plus operator,
; semicolon.

Parser :→

- Receivers token from the scanner.
- Recognize the structure of program as a parse tree.
- Pase tree is recognized a/c to a context-free grammar.
- syntax errors are reported if the program is syntactically incorrect.
- A pase tree is inefficient to represent the structure of program
- A syntax tree is a more condensed version of the pase tree.
- A syntax tree is usually generated as output by the parser.

Semantic Analyzer :→

- Semantic of program are its meaning as opposed to syntax or structure.
- The semantic consist of:
  Runtime semantics - behavior of program at runtime
  static semantics - checked by the compiler.



⇒

```
temp1 := int2real (y)
temp2 := temp1  real * 2.5
temp3 := x  real  +  temp2
a :=  temp3
```

Fig: Intermediate code Generator

Three - address code

```
temp1 := int2real(y)
temp2 := temp1  real * 2.5
temp3 := x + temp2
a := temp3
```

⇒

Assembly code (Hypothetical)

```
LOADI   R1,y      ;; R1 ← y
MOVF    F1,R1     ;; F1 ← int2real (R1)
MULF    F2,F1,2.5 ;; F2 ← F1 * 2.5
LOADF   F3,x      ;; F3 ← x
ADDF    F4,F3,F2  ;; F4 ← F3 + F2
STORF   a,F4      ;; a ← F4
```

Fig: code Generator.

Date :

- Static semantics includes:
- Declarations of variables and constants before use.
- calling functions that exists (predefined in library or defined by the user)
- Passing parameters properly.
- Type checking.
- Static semantics are difficult to check by the parser.
- Semantic analyzer does the following:
  - checks the static semantics of languages.
  - Annotates the syntax tree with type information.

Intermediate code Generation :→

- comes after Syntax and semantic analysis.
- separates the compiler front end from its backend.
- Intermediate representation should have 2 imp. properties
  - should be easy to produce
  - should be easy to translate into the target program.
- Intermediate representation can have a variety of forms:
  - Three address code, P-code for an abstract machine, Tree or DAG representation.

Code Generator :→

- Generates code for the target machine, typically:
  - Assembly code, or
  - Relocatable machine code
- Properties of the target machine become a major factor
- Code generator selects appropriate machine instruction.
- Allocates memory locations for variables.
- allocates registers for intermediate computations.

Conclusion :-

Thus, we have studies and understood the concept of compiler and phases of compiler.

Date :

Code Improvement :→

- Code improvement techniques can be applied to :
  - Intermediate code - independent of the target machine.
  - Target code - dependent on the target machine
- Intermediate code improvement includes :
- Constant folding
- Elimination of common sub-expressions
- Identification and elimination of unreachable code.
- Improving loops
- Improving function calls
- Target code improvement includes :
- Allocation and use of registers
- Selection of better (faster) instructions and addressing modes.

Conclusion :

Thus we have studies and understood the concept of compiler and phases of compiler.

Viva voce Questions :

① what is compiler ?

→ A compiler is a program that reads a program written in one language - the source language and translates it into equivalent program in another language - the target language. The compiler reports to its user the presence of errors in the source program.

② what are the two parts of the compilation ? explain briefly.

→ Analysis and Synthesis are two parts of compilation.

- Analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program

Date :

- The synthesis part constructs the desire target program from the intermediate representation.

(3) Define Symbol Table.

→ It is a data structure used by the compiler to keep track of semantics of the variables. It stores the information about scope and building info. about names.

(4) List the various phases of a compiler?

→ • lexical Analyzer
• Syntax Analyzer
• Semantic Analyzer.
• Intermediate code generator
• code optimizer
• code generator

(5) List the phases that constitute the front end of a compiler.

→ • lexical and Syntactic analysis
• The creation of Symbol table
• semantic analysis
• Generation of intermediate code.

A certain amount of code optimization can be done by front end as well. Also includes error handling that goes along with each of these phases.

(6) mention the back-end phases of compiler.

→ The backend of compiler includes those portions that depend on the target machine and generally those portion do not depend on the source language, just intermediate language. These includes.
- code optimization
- code generation, along with error handling and symbol-table operations.