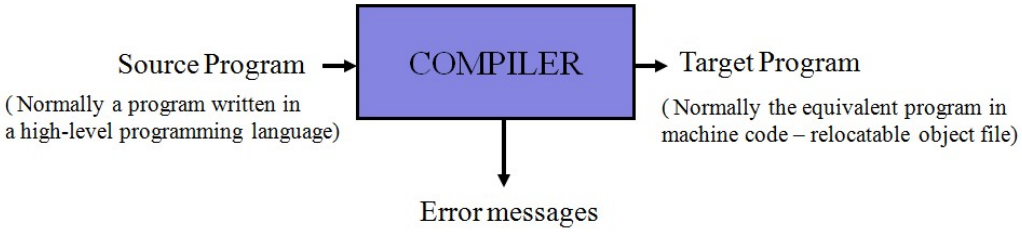
Assignment No. 1

AIM:

Introduction to Compiler.

THEORY:

A compiler is a program takes a program written in a source language and translates it into an equivalent program in a target language.



There are two parts of compilation

- Analysis part: breaks up the source program into constituents pieces and creates the intermediate representation of the source program.
  - Lexical Analyzer, Syntax Analyzer and Semantic Analyzer are the parts of this phase.
- Synthesis part: constructs the desired target program from the intermediate representation
  - Intermediate Code Generator, Code Generator, and Code Optimizer are the parts of this phase.

Language Processing System

In addition to a compiler, several other programs may be required to create an executable target program. A source program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a distinct program, called a preprocessor. The preprocessor may also expand shorthands, called macros, into source language statements.

The target program created by the compiler may require further processing before it can be run. The compiler creates assembly code that is translated by an assembler into machine code and then linked together with some library routines into the code that actually runs on the machine.

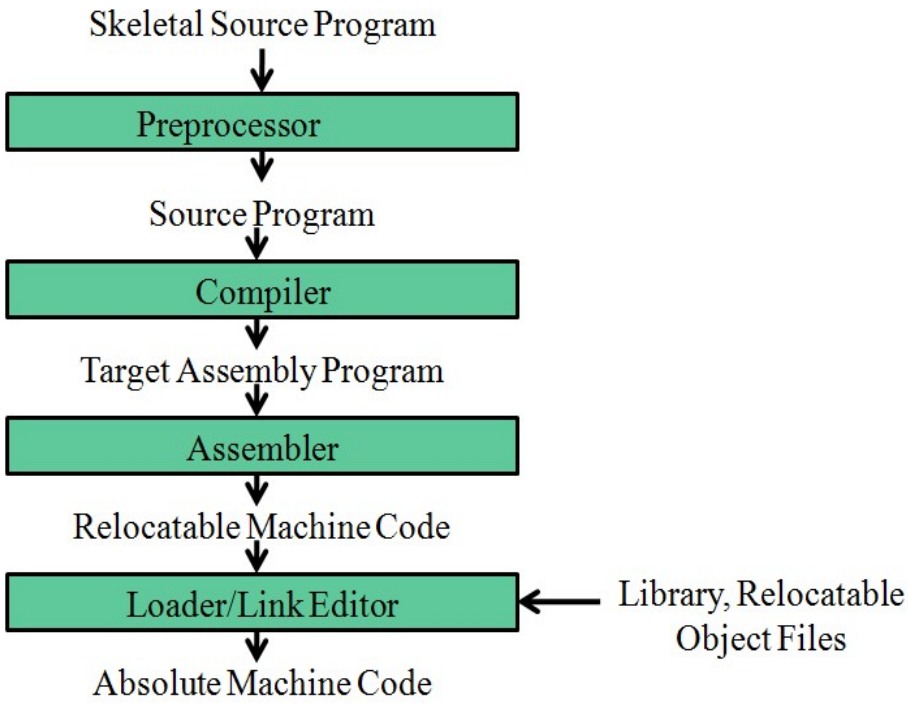


Figure: Language Processing System

Phases of Compiler:

Conceptually, a compiler operates in phases, each of which transforms the source program from one representation to another.

The first three phases form the bulk of the analysis portion of a compiler. Two other activities, symbol-table management and error handling, are shown interacting with the six phases of lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization, and code generation. Informally, we shall also call the symbol-table manager and the error handler "phases."

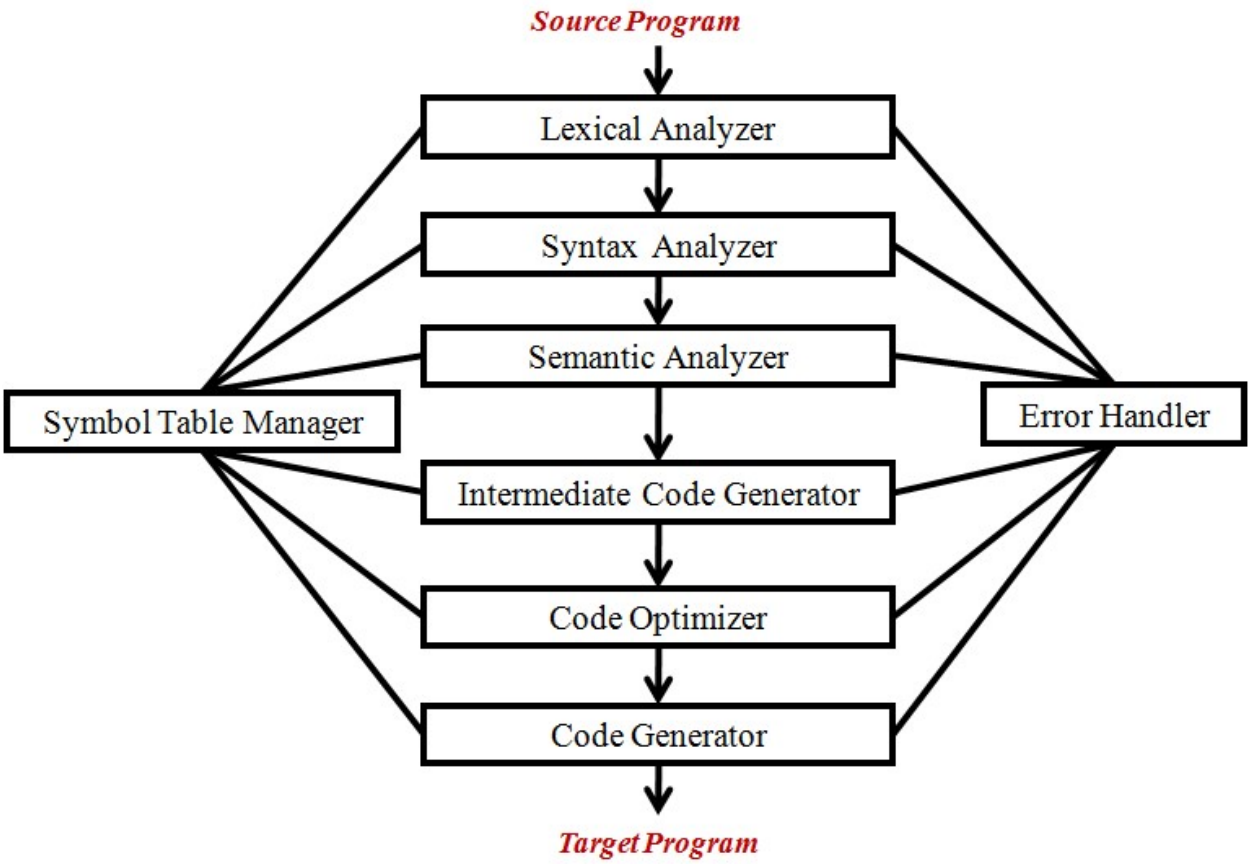


Figure: Phases of Compiler

Linear analysis: The stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning

For example, in lexical analysis the characters in the assignment statement

position := initial + rate * 60

would be grouped into the following tokens;

1. The identifier position.
2. The assignment symbol :=.
3. The identifier initial.
4. The plus sign.
5. The identifier rate.
6. The multiplication sign.
7. The number 60,

The blanks separating the characters of these tokens would normally be eliminated during lexical analysis.

Hierarchical Analysis: characters or tokens are grouped hierarchically into nested collection with collective meaning. Hierarchical analysis is called parsing or syntax analysis involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output. Usually, the grammatical phrases of the source program are represented by a parse tree

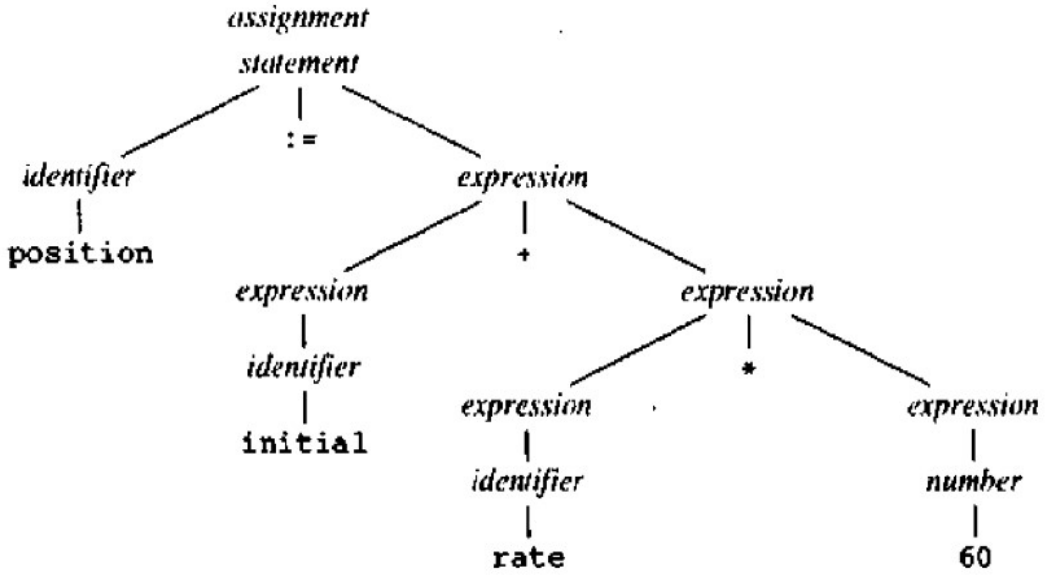
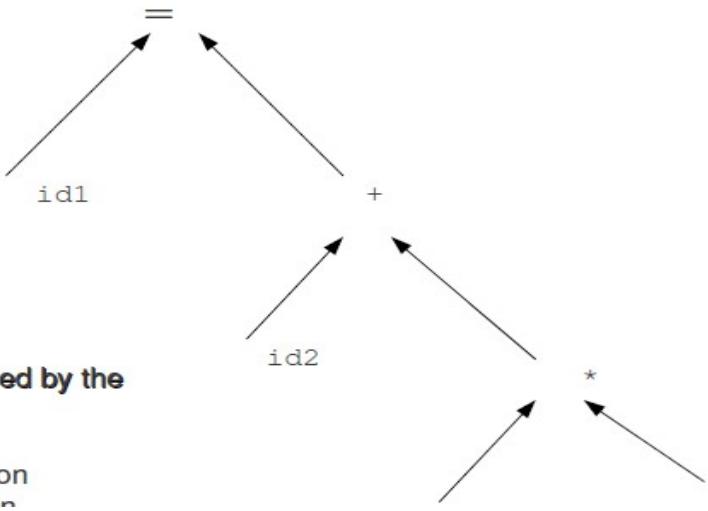


Figure: Parse Tree

A syntax tree is a compressed representation of the parse tree in which the operators appear as the interior nodes, and the operands of an operator are the children of the node for that operator.



Part of the grammar used by the syntax analysis step:

identifier is an expression

Figure: Syntax Tree

Semantic analysis: Certain checks are performed to ensure that the components of a program fit together meaningfully. A semantic analyzer checks the source program for semantic errors and collects the type information for the code generation. Type-checking is an important part of semantic analyzer. Normally semantic information cannot be represented by a context-free language used in syntax analyzers. Context-free grammars used in the syntax analysis are integrated with attributes (semantic rules). The result is a syntax-directed translation and Attribute grammars.

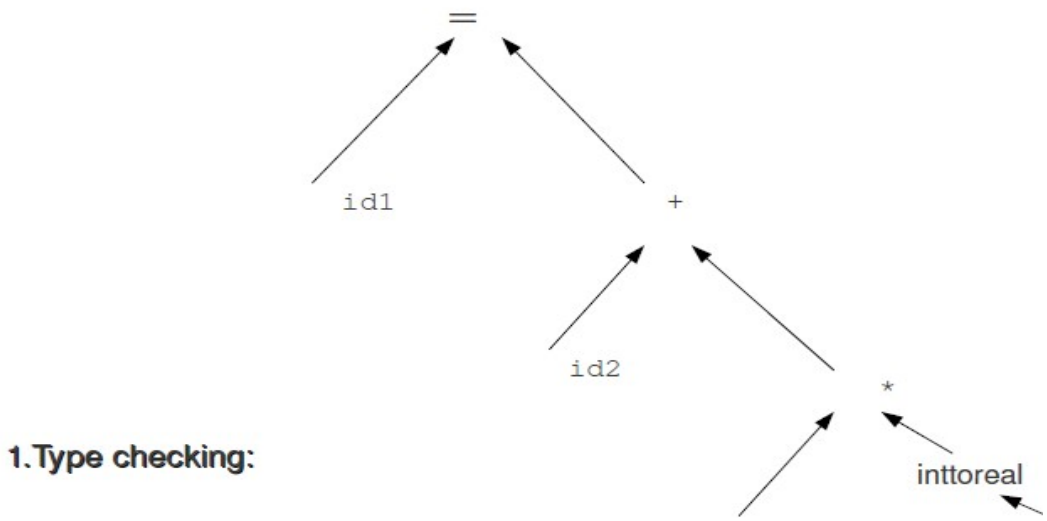


Figure: Semantic analysis inserts conversion from integer to real.

Intermediate Code Generation: A compiler may produce an explicit intermediate codes representing the source program. These intermediate codes are generally machine code (architecture independent). But the level of intermediate codes is close to the level of machine codes.

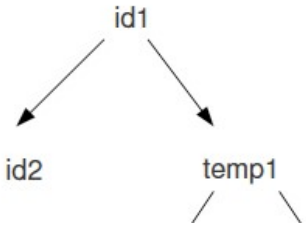
Generation of IR :

```
temp1 = inttoreal (60)
temp2 = id3 * temp1
temp3 = id2 + temp2
```

Code Optimizer (for Intermediate Code Generator): The code optimizer optimizes the code produced by the intermediate code generator in the terms of time and space.

```
temp1 = id3 * 60.0
id1 = id2 + temp1
```

or as a tree :



Code Generator: It pproduces the target language in a specific architecture. The target program is normally is a relocatable object file containing the machine codes.

assembly code or machine code for the platform:

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
```

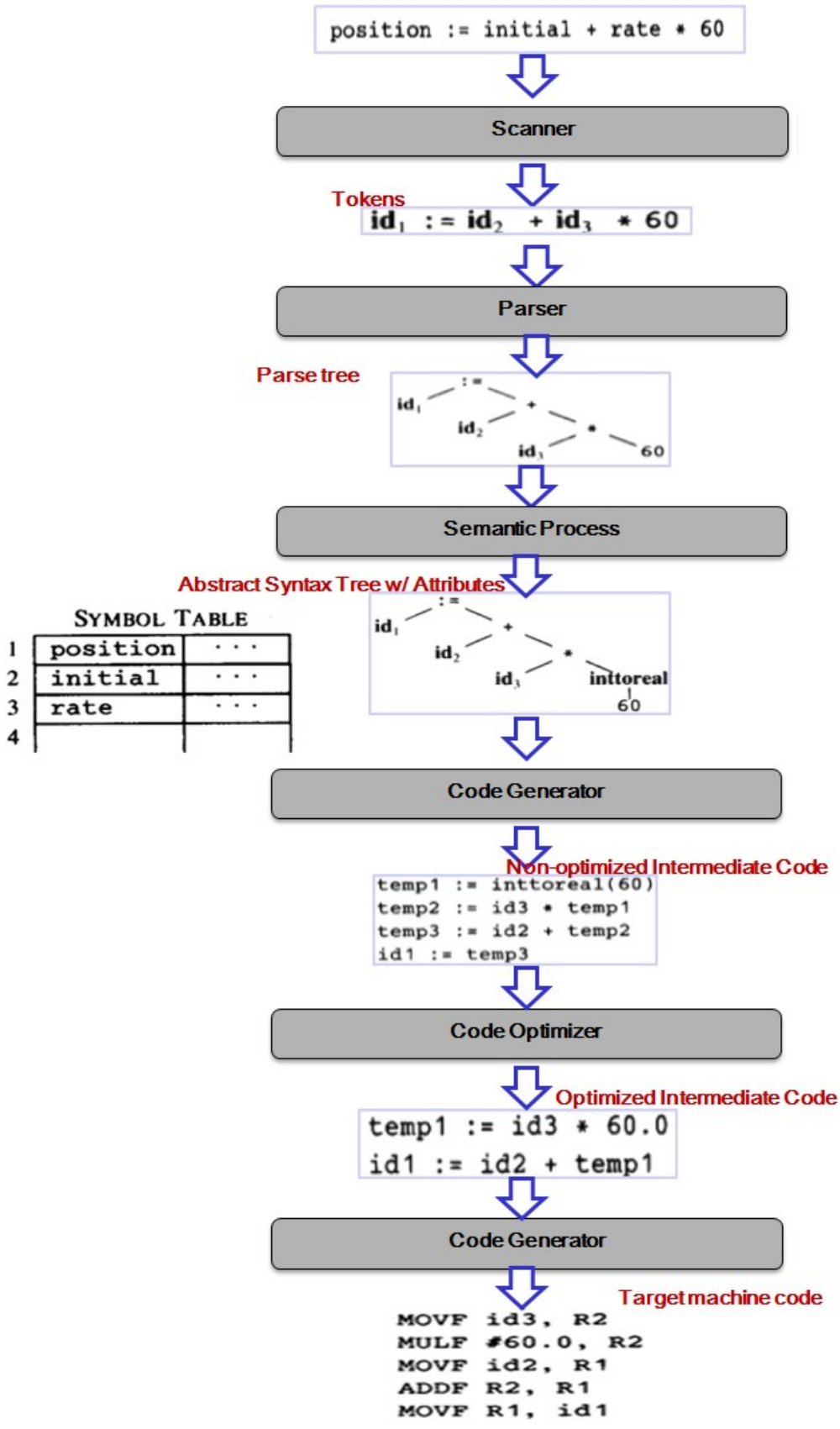
Symbol Table: An essential function of a compiler is to record the identifiers used in the source program and collect information about various attributes of each identifier. These attributes may provide information about the storage allocated for an identifier, its type, its scope (where in the program it is valid) and, in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (e.g., by reference), and the type returned, if any. A symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the

record for each identifier quickly and to store or retrieve data from that record quickly.

Error Detection and Reporting:

Each phase can encounter errors. However, after detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected. A compiler that stops when it finds the first error is not as helpful as it could be. The syntax and semantic analysis phases usually handle a large fraction of the errors detectable by the compiler.

The lexical phase can detect errors where the characters remaining in the input do not form any token of the language. Errors where the token stream violates the structure rules (syntax) of the language are determined by the syntax analysis phase. During semantic analysis the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operation involved, e. g., if we try to add two identifiers, one of which is the name of an array, and the other the name of a procedure.


position := initial + rate * 60
Scanner
Tokens
id1 := id2 + id3 * 60
Parser
Parse tree
Semantic Process
Abstract Syntax Tree w/ Attributes
SYMBOL TABLE
1 position
2 initial
3 rate
4
inttoreal
60
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 Generator
Target machine code
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1

Compiler Construction Tools

Some commonly used compiler-construction tools include

- **Parser generators:** that automatically produces syntax analyzers from a grammatical description of a programming language.
- **Scanner generators:** that produces lexical analyzers from a regular-expression description of the tokens of a language.
- **Syntax-directed translation engines:** that produces collections of routines for walking a parse tree and generating intermediate code.
- **Code-generator generators:** that produces a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.
- **Data-flow analysis engines:** that facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data-flow analysis is a key part of code optimization.

CONCLUSION:

A compiler is a program takes a program written in a source language and translates it into an equivalent program in a target language. There are two parts of compilation

- Analysis part.
  - Lexical Analyzer, Syntax Analyzer and Semantic Analyzer are the parts of this phase.
- Synthesis part.
  - Intermediate Code Generator, Code Generator, and Code Optimizer are the parts of this phase.

REFERENCES:

- Compilers - Principles, Techniques and Tools - A.V. Aho, R. Shethi and J. D. Ullman (Pearson Education)