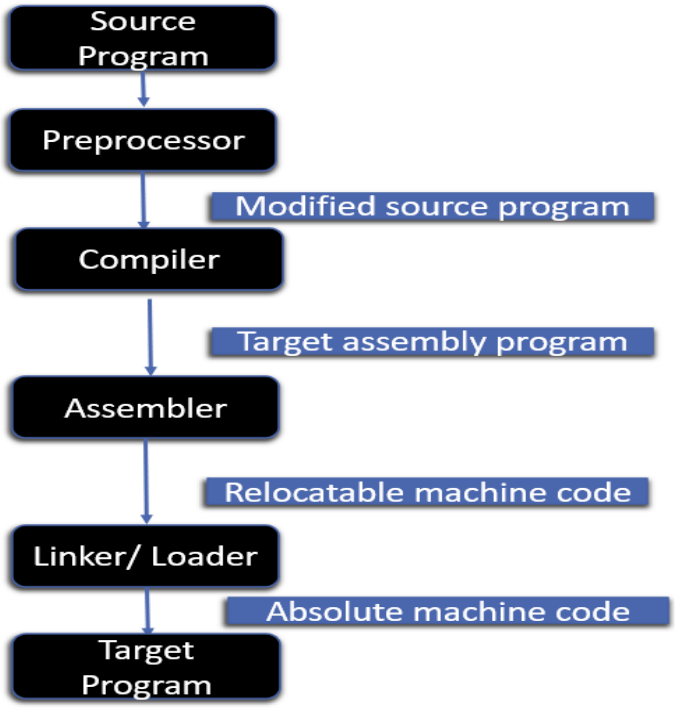
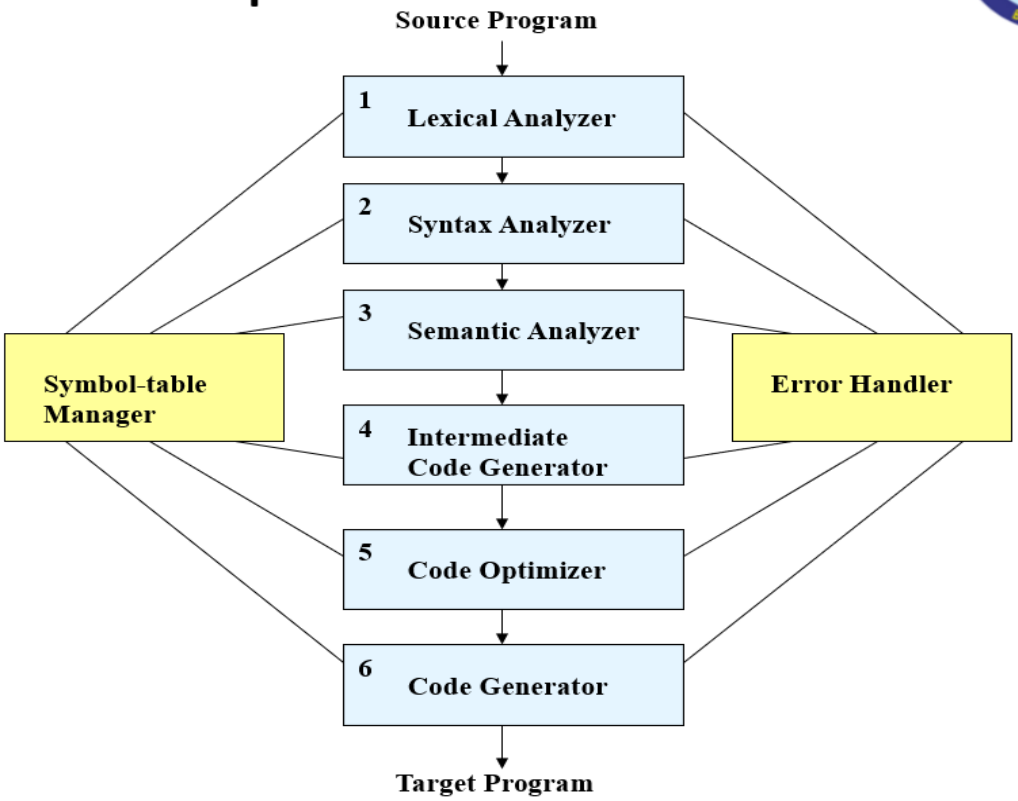
Language Processing System



The Phases of a Compiler



Two main Phases of a Compiler

1. **Analysis Phase**: Breaks up a source program into constituent pieces and produces an internal representation of it called intermediate code.
    I. Lexical Analyzer
    II. Syntax Analyzer
    III. Semantic Analyzer
    IV. Intermediate code generator
2. **Synthesis Phase**: Translates the intermediate code into the target program.
    V. Code optimizer
    VI. Code generator

Lexical Analyzer: The first phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token of the form *(token-name, attribute-value).* For example, suppose a source program contains the assignment statement

p o s i t i o n = i n i t i a l + r a t e * 60

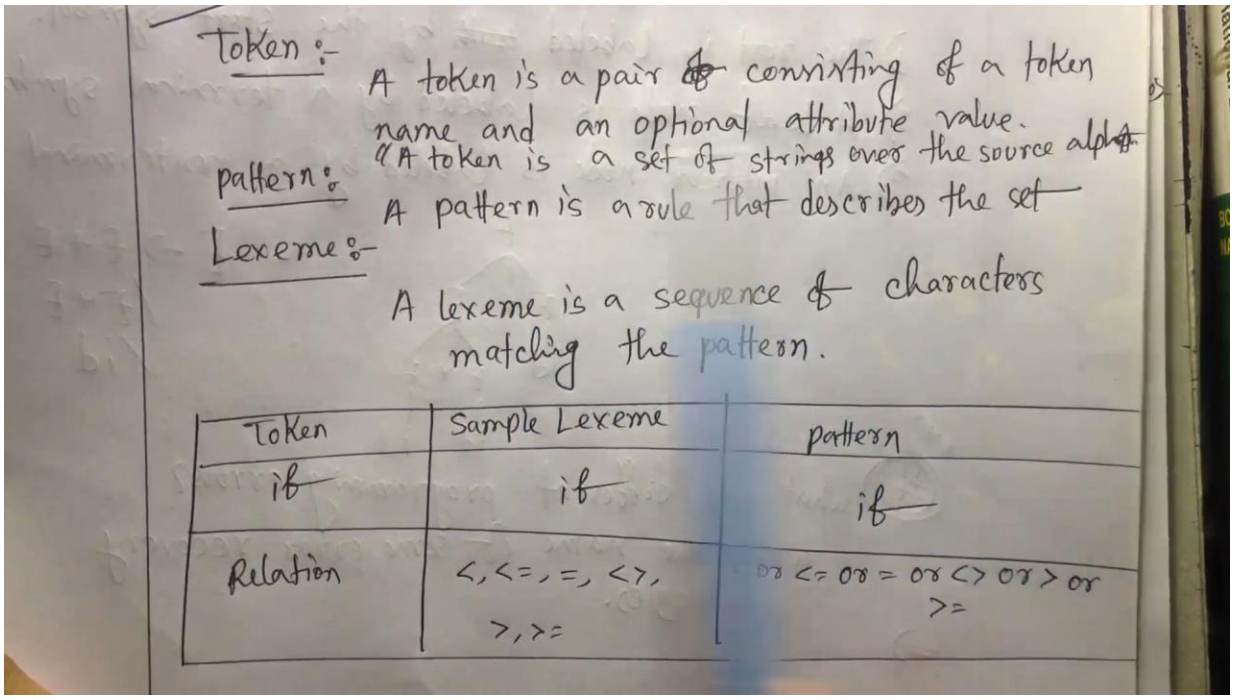
The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:

1. p o s i t i o n is a lexeme that would be mapped into a token **(id,** 1), where **id** is an abstract symbol standing for *identifier* and 1 points to the symbol table entry for p o s i t i o n. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.
2. The assignment symbol = is a lexeme that is mapped into the token (=). Since this token needs no attribute-value, we have omitted the second component. We could have used any abstract symbol such as **assign** for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.
3. Initial is a lexeme that is mapped into the token **(id,** 2), where 2 points to the symbol-table entry for i n i t i a l .
4. + is a lexeme that is mapped into the token (+).

5. r a t e is a lexeme that is mapped into the token **(id,** 3), where 3 points to the symbol-table entry for r a t e.
6. * is a lexeme that is mapped into the token (*).
7. 60 is a lexeme that is mapped into the token (60).

After lexical analyzer compiled the above expression the output of lexical analyzer would be

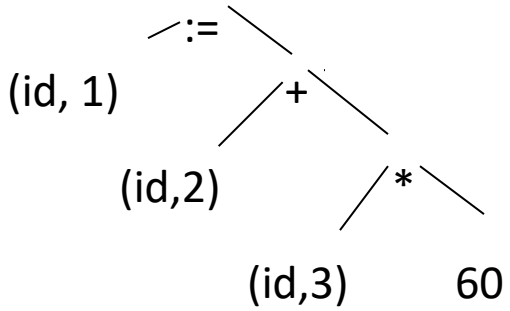
<i d , l > <=> <id, 2> <+> **<id,** 3> <*> <60>



Lexical Errors: A lexical error is a mistake in a lexeme, for examples, typing **tehn** instead of **then**, or missing off one of the quotes in a literal.

Syntax Analyzer: The second phase of the compiler is syntax analysis or parsing. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the

arguments of the operation. A syntax tree for the token stream (obtain from lexical analyzer) is shown below as the output of this phase.

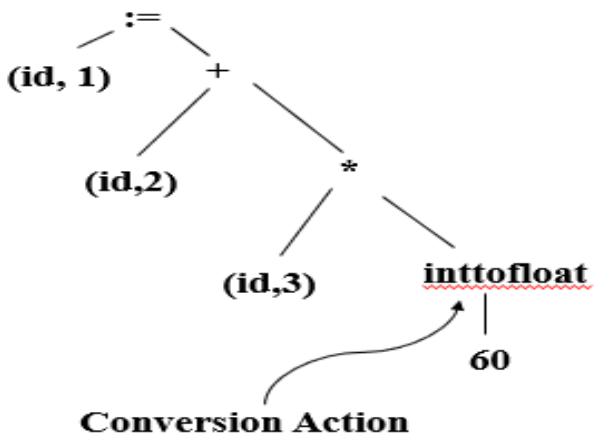


The tree has an interior node labeled * with **(id, 3)** as its left child and the integer **60** as its right child. The node **(id, 3)** represents the identifier **rate.** The node labeled * makes it explicit that we must first multiply the value of **r a t e** by **60.** The node labeled + indicates that we must add the result of this multiplication to the value of i n i t i a l . The root of the tree, labeled =, indicates that we must store the result of this addition into the location for the identifier p o s i t i o n . This ordering of operations is consistent with the usual conventions of arithmetic which tell us that multiplication has higher precedence than addition, and hence that the multiplication is to be performed before the addition.

Syntax Error: A grammatical error is a one that violates the (grammatical) rules of the language, for example if x = 7 y := 4 (missing **then**).

Semantic Analyzer: The *semantic analyzer* uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. In practice semantic analyzers are mainly concerned with type checking and type coercion based on type rules. The semantic analyzer produces an annotated syntax tree as an output.



Semantic Errors: During compilation Semantic analyzer will recognize the following semantic errors.

- ❖ Datatype mismatch
- ❖ Undeclared variable
- ❖ Multiple declaration of a variable in a scope
- ❖ Actual and formal parameter mismatch