UNIT-1

INTRODUCTION

1.1 Language Processors:

A ***Compiler*** is a program that can read a program in one language, the source language and translate it into an equivalent program in another language, the target language.

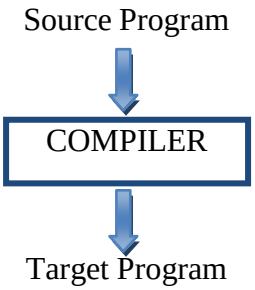


Fig 1.1 A Compiler

An important role of the compiler is to report any errors in the source program that it detects during the translation process.

Fig 1.2 Running the Target Program

If the target program is an executable machine language program, it can then be called by the user to process inputs and produces outputs.

An ***Interpreter*** is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.

Fig 1.3 An Interpreter

- The Machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs.
- An Interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

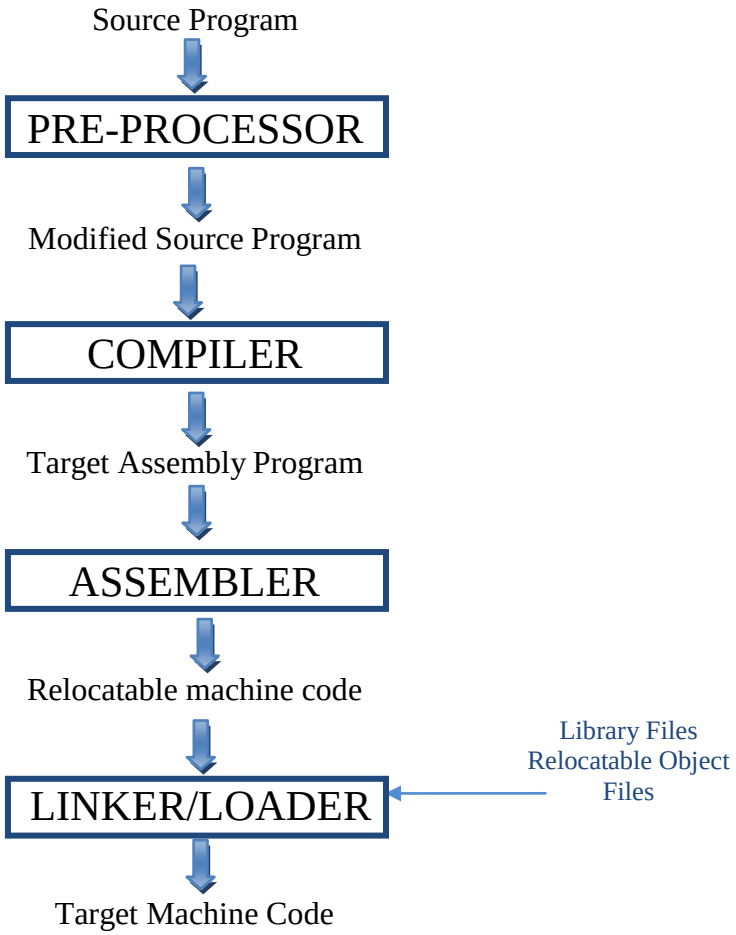


Fig 1.4 A Language-files Processing Systems

1.2 The Structure of a Compiler.

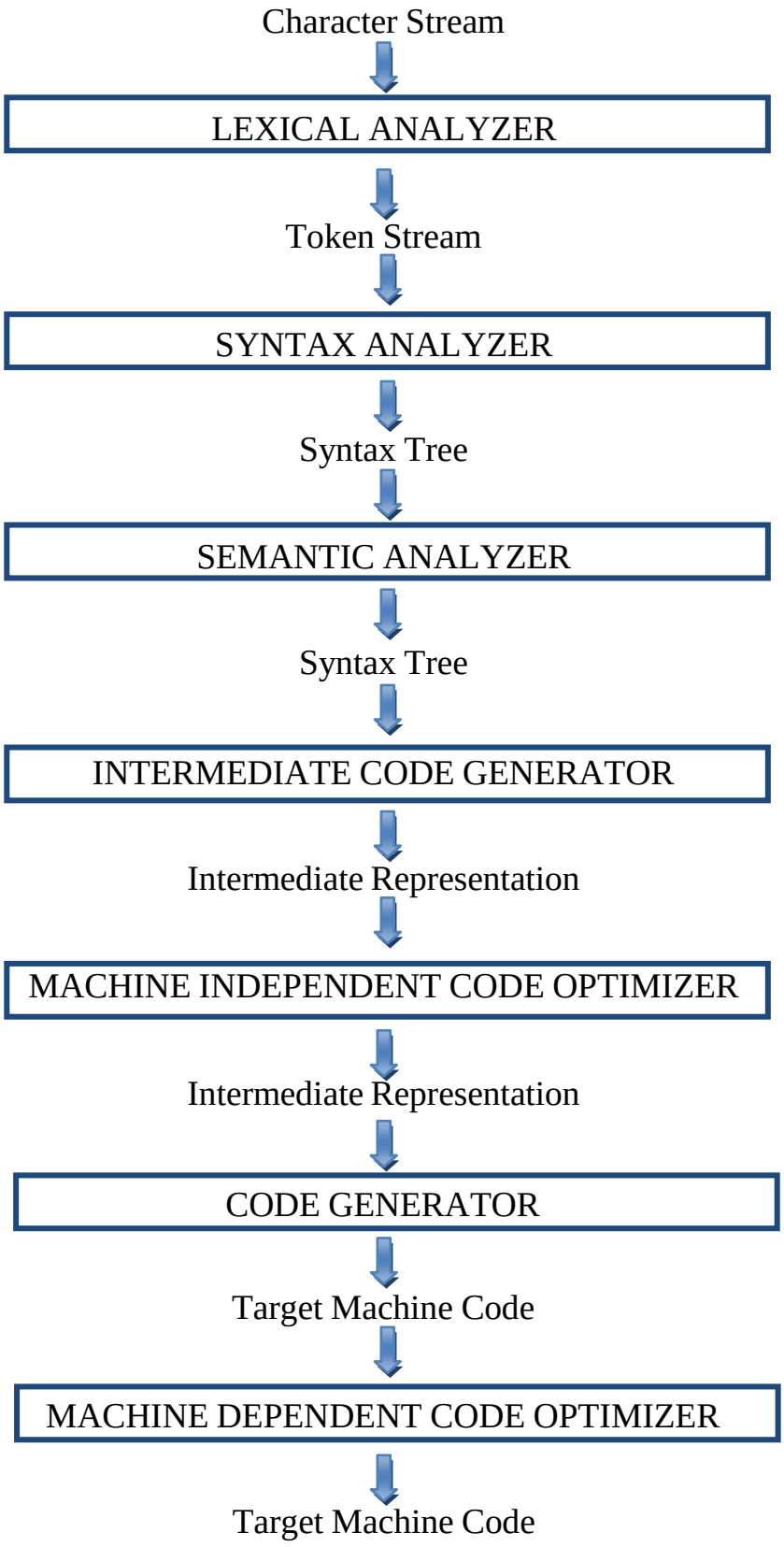


Fig 1.5 Phase of a Compiler

Compiler maps a source program into a semantically equivalent target program.

There are two parts of the mapping:

1. Analysis
2. Synthesis

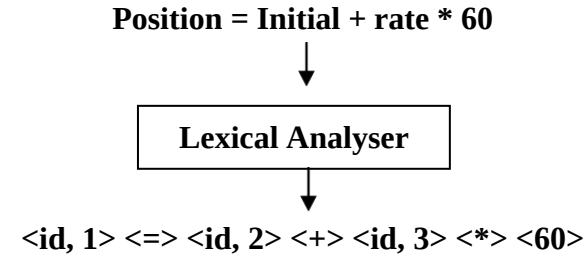
Analysis Part

- The analysis part break up the source program into constituent pieces and implies a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program.
- If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative message, so the user can take corrective action.
- The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.

Synthesis Part

- The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table.
- The analysis part is often called the front end of the compiler and the synthesis part is the back end.

1.2.1 Lexical Analysis



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 sequence called *lexemes*.
For each lexeme, the lexical analyzer produces as output a token of the form:

<token-name, attribute-value>

In the token, the first component token-name is an abstract symbol that is used during syntax analysis and the second component attribute-values points to an entry in the symbol table for this token.

Position =initial + rate *60

1. *Position* is a lexeme that would be mapped into a token ***<id, 1>***, where *'id'* is an abstract symbol standing for an identifier and ***1*** points to the symbol table entry for position
2. The assignment symbol = is a lexeme that is mapped into the token <=>
3. *Initial* is a lexeme that is mapped into the token ***<id, 2>,*** where ***2*** points to the symbol table entry for initial.

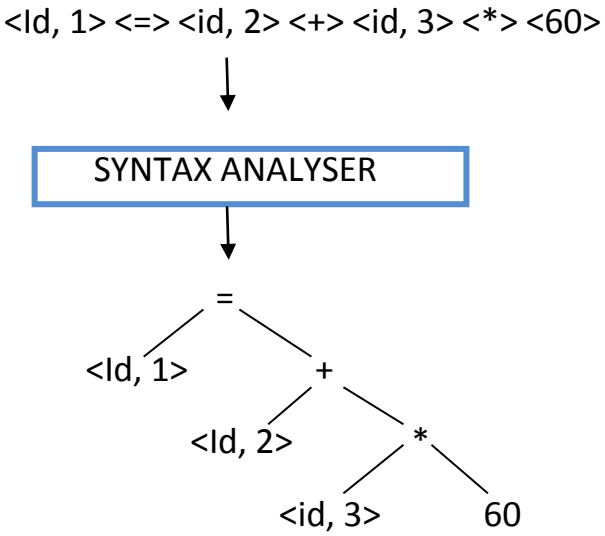
4. + is a lexeme that is mapped into the token < + >
5. *Rate* is a lexeme mapped into the token <***id, 3***>, where ***3*** points to the symbol table entry for rate
6. ******* is a lexeme mapped into the token < ******* >
7. ***60*** is a lexeme mapped into the token < ***60*** >

Blanks separating the lexeme would be discarded by the lexical analyzer.

1.2.2 Syntax Analysis

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 tree-like intermediate representation that depicts the grammatical structure of the token stream.

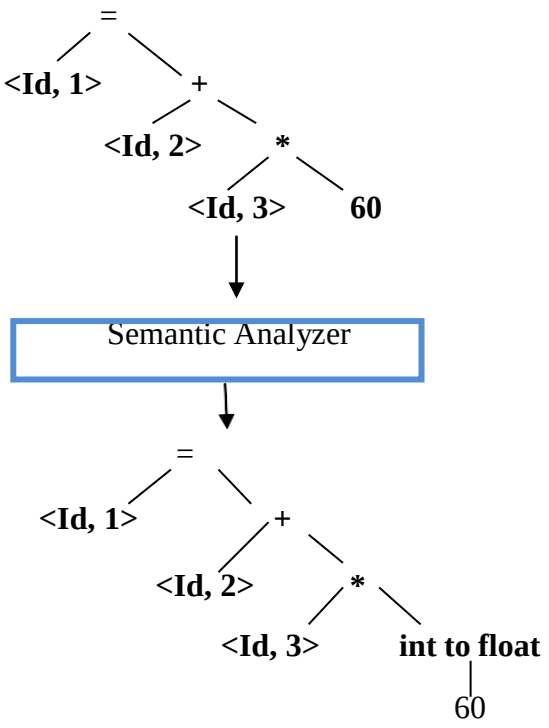
A Typical representation is a syntax tree in which each interior node represents an operation and the children of the node represents the arguments of the operation.



The tree has an interior node labeled * with <id, 3> as its left child and the integer 60 as is right child. The node <id, 3> represents the identifier rate. The node labeled * makes it explicitly that are must first multiply the value of rate by 60.The node labeled + indicates that are must add the result of this multiplication to the value of initial.

- The root of the tree, labeled = indicates that are must store the result of this addition into the location for the identifier position.
- This ordering of operation is considered 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.

1.2.3 Semantic Analysis



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.

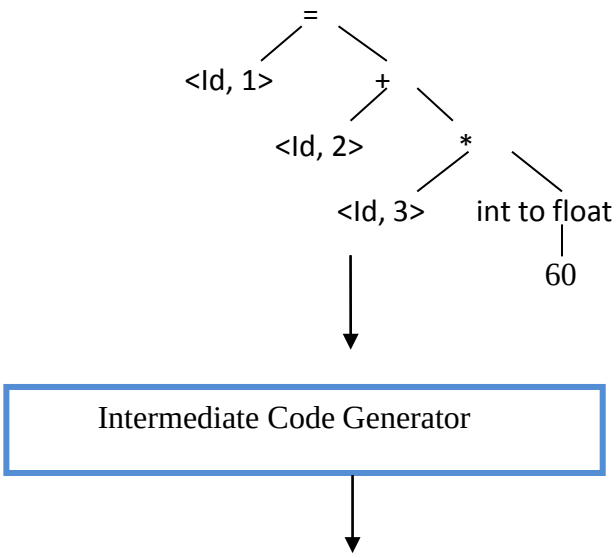
An important part of semantic analysis is type checking, where compiler checks that each operator has matching operands.

For example: Many programming language definitions require an array index to be an integer. The compiler must report an error if a floating point number is used to index an array.

The language specification may permit some type conversion called Coercions.

For example, a binary arithmetic operator may be applied to either a pair of integer or to a pair of floating point number. If the operator is applied to a floating point number and an integer, the compiler may convert the integer into a floating point number.

1.2.4 Intermediate Code Generation.



t1= int to float (60)

t2=id3*t1

t3=id2+t2

id1=t3

In the process of translating source program into target code, a compiler may construct one or more intermediate representation.

This intermediate representation should have two important properties.

- It should be easy to produce
- It should be easy to translate into the target machine

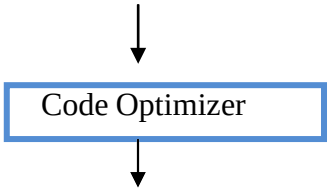
1.2.5 Code Optimization

t1 = int to float (60)

t2 = id3 * t1

t3 = id2 + t2

id1 = t3



t1 = id3 * 60.0

id1 = id2 + t1

The machine-independent code-optimization phase attempts to improve the intermediate code so that faster target code will result.

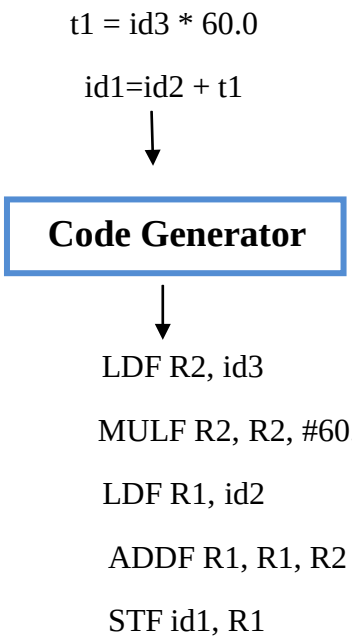
A simple intermediate code generation algorithm followed by code optimization is a reasonable way to generate good target code. The optimizer can deduce that the conversion of 60 from integer to floating point can be done once and for all at compile t ime, so that int to float operation can be eliminated by replacing the integer 60 by the floating point number 60.0.

t1 = id3 *60.0

id1 = id2 + t1

There is a great variation in the amount of code optimization different compilers perform.

1.2.6 Code Generation



1.5

The code generator takes an input as intermediate representation of the source program and maps it into the target language. If the target language is machine code, register or memory locations are selected for each of the variable used by the program. Then, the intermediate instructions are translated into sequence of machine instructions that perform the same task.

The First operand of each instruction specifies a destination. The F in each instruction tells us that it deals with floating point numbers. The code in (1.5) loads the content of address id3 into register R2. Then multiply it with floating-point constant 60.0.

1.2.7 Symbol-Table Management

1. Position	….
2. Initial	….
3. Rate	….

The Symbol table is a data structure containing a record for each variable name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.