**1.0 COMPILERS**

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

****

**1.1 PARTS OF COMPILATION**

There are two parts to compilation:

1. analysis
2. synthesis

The ***analysis part*** breaks up the source program into constituent pieces and creates an intermediate representation of the source program.

The ***synthesis part*** constructs the desired target program from the intermediate representation. Of the two parts, synthesis requires the most specialized techniques.

During analysis, the operations implied by the source program are determined and recorded in a hierarchical structure called a ***tree***.

Often, a special kind of tree called a ***syntax tree*** is used, in which each node represents an operation and the children of a node represent the arguments of the operation.



Many software tools that manipulate source programs first perform some kind of analysis. Some examples of such tools are

* Structure editor
* Pretty printers
* Static checkers
* interpreters

**Structure editor**

* A structure editor takes as input a sequence of commands to build a source program.
* The structure editor not only performs the text-creation and modification functions of an ordinary text editor, but it also analyzes the program text, putting an appropriate hierarchical structure on the source program.
* For example, it can check that the input is correctly formed, can supply keywords automatically (e-g.. when the user types while. the editor supplies the matching do and reminds the user that a conditional must come between them), and can jump from a begin or left parenthesis to its matching end or right parenthesis.

**Pretty printers**

* A pretty printer analyzes a program and prints it in which a way that the structure of the program becomes clearly visible.
* For example, comments may appear in a special font, and statements may appear with an amount of indentation proportional to the depth of their nesting in the hierarchical organization of the statements.

**Static checkers**

* A static checker reads a program, analyzes it, and attempts to discover potential bugs without running the program.
* For example, a static checker may detect that parts of the source program can never be executed.
* It can catch logical errors such as trying to use a real variable as a pointer.

**Interpreters**

* Interpreter performs the operations implied by the source program.
* For an assignment statement, for example, an interpreter might build a tree like Fig. 1.2, and then carry out the operations at the nodes as it "walks" the tree.
* Interpreters are frequently used to execute command languages, since each operator executed in a command language is usually an invocation of a complex routine such as an editor or compiler.
* The analysis portion in each of the following examples is similar to that of a conventional compiler.

**1.3 ANALYSIS OF THE SOURCE PROGRAM**

Analysis consists of **three parts**

i) **Linear Analysis** - is called lexical analysis or scanning. It is the process of reading a character from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.

ii) **Hierarchical analysis –** is called as syntax analysis or parsing. In this analysis the characters or tokens are grouped hierarchically into nested collections with collective meaning.

iii)**Semantic analysis** - in which certain checks are performed to ensure that the components of a program fit together meaningfully. i.e it check the source program for semantic errors and gathers type information for subsequent code generation phase.

**THE PHASES OF A COMPILER**

* A compiler operates in phases, each of which transforms the source program from one representation to another.
* A typical decomposition of a compiler is shown in Fig, 1.9,
* The first three phases forming the bulk of the analysis portion of a compiler.
* Symbol - table management and error handling, are shown interacting with the six phases of the compiler.

**SYMBOL-TABLE MANAGEMENT**

* 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 and, in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument and the type returned.
* 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.
* When an identifier in the source program is detected by the lexical analyzer, the identifier is entered into the symbol table.
* However, the attributes of an identifier cannot normally determined during lexical analysis. For example, in a Pascal declaration like

**var position, initial , rate : real ;**

* the type real is not known when position, initial , and rate are seen by the lexical analyzer.

The remaining phases enter information about identifiers into the symbol table and then use this information in various ways.



**ERROR DETECTION AND REPORTING**

* Each phase can encounter errors. However, after detecting an error, a phase must deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected.
* 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 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.

**THE ANALYSIS PHASES - LEXICAL ANALYSIS**

* The lexical analysis phase reads the characters in the source program and groups them into a stream of tokens.
* Each token represents a logically cohesive sequence of characters, such as an identifier, a keyword (if, while, etc,), a punctuation character, or a multi-character operator like :=.
* The character sequence forming a token is called the lexeme for the token.
* Certain tokens will be augmented by a "lexical value."
* The lexical analyzer not only generates a token, say id, but also enters the lexeme rate into the symbol table.
* Consider the above expression

position := initial + rate \* 60

* The representation of the above expression after the lexical analysis is

id1 = id2 + id3 \* 60

**SYNTAX ANALYSIS**

* It groups token together into syntactic structures. (Fig.1.11a.. syntax tree)
* A typical data structure for the tree is shown in Fig. 1.11(b) in which an interior node is a record with a field for the operator and two fields containing pointers to the records for the left and right children.
* A leaf is a record with two or more fields, one to identify the token at the leaf, and the others to record information about the token.



**SEMANTIC ANALYSIS**

* An important component of semantic analysis is type checking. Here the compilers checks that each operator has operands that are permitted by the source language specification.
* For eg. Many programming language definition require a compiler to report an error every time a real number is used to index an array.
* However, the language specification may permit some operand coercions, for example, when a binary arithmetic operator is applied to an integer and real, in this case, the compiler may need to convert the integer to a real.

**INTERMEDIATE CODE GENERATION**

* After syntax and semantic analysis, some compilers generate an explicit intermediate

representation of the source program.

* This intermediate representation should have two important properties. it should be easy to produce and easy to translate into the target program.
* The intermediate representation can have a variety of forms and one of the forms is called “Three address code”, which is like the assembly language for a machine in which every memory location can act like a register.
* Three-address code consists of a sequence of instructions, each of which has at most three operands.
* Three address code for the statement **position : = initial + rate \* 60** is



**Inter mediate form has several properties.**

* **First,** each three-address instruction has at most one operator in addition to the assignment. Thus, when generating these instructions, the compiler has to decide on the order in which operations are to be done; the multiplication precedes the addition in the source program of (1.1).
* **Second,** the compiler must generate a temporary name to hold the value computed by each instruction.
* **Third,** some "three address" instructions have fewer than three operands, e.g., the first and last instructions in ( 1.3).

**CODE OPTIMIZATION**

* The code optimization phase attempts to improve the intermediate code, so that faster-running machine code will result.
* The above intermediate code is optimized like this,



* inttoreal operation can be eliminated by the conversion of 60 integer in to real and temp3 is used only once, to transmit its value to id1,so it can be eliminated.

**CODE GENERATION**

* The final phase of the compiler is the generation of target code, consisting normally of relocatable machine code or assembly code, Memory locations are selected for each of the variables used by the program.
* Then, intermediate instructions are each translated into a sequence of machine instructions that perform the same task.
* A crucial aspect is the assignment of variables to registers. For example, using registers 1and 2, the translation of the code of the above code might become



* The first and second operands of each instruction specify a source and destination, respectively. The F in each instruction tells us that instructions deal with floating-point numbers.
* This code moves the contents of the address id3 into register 2, then multiplies it with the real constant 60.0
* The third instruction moves id2 into register 1 and adds to it the value previously computed in register 2. Finally, the value in register 1 is moved into the address of id1.



**COUSINS OF THE COMPILER**

The input to a compiler may be produced by one or more preprocessors, and further processing of the compiler's output may be needed before running machine code is obtained.

**Preprocessors**

Preprocessors produce input to compilers. They may perform the following functions:

1. Macro processing - A preprocessor may allow a user to define macros that are short hands for longer constructs.
2. File inclusion - A preprocessor may include header files into the program text. For example, the C preprocessor causes the contents of the file <global.h> to replace the statement #include <global.h> when it processes a file containing this statement.
3. "Rational" preprocessors - These processors augment older languages with more modern flow-of-control and data-structuring facilities. For example, such a preprocessor might provide the user with built-in macros for constructs like while-statements or if-statements, where none exist in the programming language itself.
4. Language extensions, These processors attempt to add capabilities to the language by what amounts to built-in macros, For example. the language Equel is a database query language embedded in C.

Statements beginning with ## are taken by the preprocessor to be database-access statements, unrelated to C, and are translated into procedure calls on routines that perform the database access.

Macro processors deal with two kinds of statement:

* macro definition
* macro use

Definitions are normally indicated by some unique character or keyword, like define or macro. They consist of a name for the macro being defined and a body, forming its definition.

The use of a macro consists of naming the macro and supplying actual parameters, that is

Values for its formal parameters. The macro processor substitutes the actual parameters for the formal parameters in the body of the macro; the transformed body then replaces the macro use itself.

**Assemblers**

* Some compilers produce assembly code that is passed to an assembler for further processing, Other compilers perform the job of the assembler, producing relocatable machine code that can be passed directly to the loader/link-editor.
* Assembly code is a mnemonic version of machine code, in which names are used instead of binary codes for operations, and names are also given to memory addresses.
* A typical sequence of assembly instruction might be



This code moves the contents of the address a in to register 1, then adds the constant 2 to it and finally stores the result in the location named by b. Thus, it computes b : = a + 2.

**Two pass assembly**

* The simplest form of assembler makes two passes over the input, where a pass consists of reading an input file once. In the first pass, all the identifiers that denote storage locations are found and stored in a symbol table.
* Identifiers are assigned storage locations as they are encountered for the first time, so after reading, the symbol table might contain the entries shown in Figure. In that figure, we have assumed that a word, consisting of four bytes, is set aside for each identifier, and that addresses are assigned starting from byte 0.

**Symbol table**



* In the second pass, the assembler scans the input again. This time, it translates each operation code into the sequence of bits representing that operation in machine language, and it translates each identifier representing a location into the address given for that identifier in the symbol table.
* The output of the second pass is usually relocatable machine code, meaning that it can be loaded starting at any location L in memory.

**Loaders and link-editors**

* Usually, a program called a loader performs the two functions of loading and link-editing.
* The process of loading consists of taking relocatable machine code, altering the relocatable addresses, and placing the altered instructions and data in memory at the proper locations.
* The link-editor allows us to make a single program from several files of relocatable machine code, These files may have been the result of several different compilations, and one or more may be library files of routines provided by the system and available to any program that needs them.
* If the files art to be used together in a useful way, there may be some external references, in which the code of one file refers to a location in another file. This reference may be to a data location defined in one file and used in another, or it may be to the entry point of a procedure that appears in the code for one file and is called from another file.
* The relocatable machine code file must retain the information in the symbol table for each data location or instruction label that is referred to externally. If we do not know in advance what might be referred to, we in effect must include the entire assembler symbol table as part of the relocatable machine code.

**COMPILER-CONSTRUCTION TOOLS**

In addition to these software-development tools, .other more specialized tool has been developed for helping implements various phases of a compiler.

Some general tools have been created for the automatic design of specific compiler components, these tools use specialized languages for specifying and implementing the component, and many use algorithms that are quite sophisticated.

The following is a list of some useful compiler- construction tools:

1. **Parser generators** - These produce syntax analyzers, normally from input that is based on a context-free grammar. In early compilers, syntax analysis consumed not only a large fraction of the running time of a compiler, but a large fraction of the intellectual effort of writing a compiler. This phase is now considered one of the easiest to implement. Many parser generators utilize powerful parsing algorithms that are too complex to be carried out by hand.
2. **Scanner generators -** These automatically generate lexical analyzers, normally from a specification based on regular expressions. The basic organization of the resulting lexical analyzer is in effect a finite automaton.
3. **Syntax-directed translation engines** - These produce collections of routines that walk the parse tree, such as intermediate code. The basic idea is that one or more "translations" are associated with each node of the parse tree, and each translation is defined in terms of translations at its neighbor nodes in the tree.
4. **Automatic code generators** - Such a tool takes a collection of rules that define the translation of each operation of the intermediate language into the machine language for the target machine.

The rules must include sufficient detail that we can handle the different possible access methods for data; e.g.. Variables may be in registers, in a fixed (static) location in memory, or may be allocated a position on a stack.

The basic technique is "template matching." The intermediate code statements are replaced by "templates" that represent sequences of machine instructions, in such a way that the assumptions about storage of variables match from template to template.

1. **Data flow engines -** Much of the information needed to perform goodcode optimization involves "data-flow analysis," the gathering of information about how values are transmitted from *one* part of a program to each other part. Different tasks of this nature can be performed by essentially the same routine, with the user supplying details of the relationship between intermediate *code* statements and the information being gathered.