**INTRODUCTION TO COMPILER**

**Inside this Chapter**

* 1. Introduction to Programming Languages
  2. High Level Programming Languages
  3. The Structure of the Program
  4. Introduction of Translators
  5. Why Study Compilers
  6. What is the Challenge in Compiler Design?
  7. Compiler Requirements
  8. The Components of a Compiler
  9. The Number of Compiler Passes
  10. Introduction to Bootstrapping (Compilers)
  11. The Target Architecture of Compiler
  12. Language Implementation and Compiler Design

**1.1 Introduction to Programming Languages**

We can easily analyze it, that every concept of designing of computer software is inspired by the life. The same thought is applied in the development of programming languages. Like Natural Languages (Hindi, English, …) programming languages are also notations, with which common man can communicate to the computers.

Coded language used by programmers to write instructions that a computer can understand to do what the programmer (or the computer user ) wants. The most basic (called low-level) computer language is the machine language that uses binary (‘1’ and ‘0’) code which a computer can run (execute) very fast without using any translator or interpreter program, but is tedious and complex. The high-level languages (such as Basic, C, Java) are much simpler (more ‘English-like’) to use but need to use another program (a compiler or an interpreter) to convert the high-level code into the machine code, and are therefore slower. There are dozens of programming languages and new ones are being continuously developed. Also called computer language.

According to **Nico Habermann** “An ideal language allows us to express what is useful for the programming task and at the same time makes it difficult to write what leads to incomprehensible or correct programs”.

According to **Robert Harper** “Good language makes it easier to establish, verify and maintain the relationship between code and its properties”.

**1.1.1 The History of Computer Programming Languages**

In 1945, John Von Neumann developed two important concepts that directly affected the path of computer programming languages. The first was known as should be ‘shared program technique’. The technique states that the actual computer hardware should be simple and need not to be hand-wired for each program. Instead, complex instructions should be used to control the hardware, allowing it to be reprogrammed much faster. The second concept was extremely important for development of programming languages. Von Neumann called it “Conditional Control Transfer”. This idea gave rise to the notation of subroutines, or small block of code that could be jumped to in any order, instead of a single set of chronologically ordered steps for the computer to take.

In 1949, a few years, after Von Neumann’s work, the language short code appeared. It was the first computer language for electronic devices and it required the programmer to change its statements in to 0’s and 1’s by hand. In 1957, the first of the major languages appeared in the form of FORTRAN. Its name stands for FORmula TRANslating system. The language was designed at IBM for scientific computing. Though FORTRAN was good at handling numbers, it was not good at handling input and output, which mattered most to business computing. Business computing started to take off in 1959 and because of this COBOL was developed. COBOL statements also have a very English-like grammar, making it quite easy to learn. All of these features were designed to make it easier for the average business to learn and adopt it.

In 1958, John McCarthy of MIT created the LIST Processing (or LISP) language. It was designed for Artificial Intelligence (AI) research. Because it was designed for such a highly specialized field, its syntax has rarely seen before or since. The Algol language was created by a committee for scientific use in 1958. Its major contribution is being the root of the tree that has led to such languages as Pascal, C, C++ and Java. It was also the first language with a formal grammar, known as Backus-Naur Form or BNF. Pascal was begun in 1968 by Niklaus Wirth. Its development was mainly out of necessity for a good teaching tool. Wirth later created a successor to Pascal, Modula-2, but by the time it appeared, C was gaining popularity and users at a rapid pace. C was developed in 1972 by Dennis Ritchie while working at Bell Labs in New Jersey. In the late 1970’s and early 1980’s, a new programming method was being developed. It was known as object oriented programming, OOP. Bjarne developed extensions to C known as “C with classes”. This set of extensions developed into the full-feature language C++, which was released in 1983.

In 1994, the Java project team changed their focus to the web. The next year, Netscape licensed Java for use in their internet browser, Navigator. In 1964 BASIC was developed by the John Kemeny and Thomas Kurtz. Microsoft has extended BASIC in its Visual Basic (VB) product. In 1987 Perl was developed by Larry Wall, it has very strong text matching functions which make it ideal for gateway designing for a web interface.

Programming language have been under development for years and will remain so for many years to come. They got their start with a list to wire a computer to perform a task. These steps eventually found their way into software and began to acquire newer and better features. The first major languages were characterized by the simple fact that they were intended for one purpose only, while the languages of today are differentiated by the way they are programmed in, as they can be used almost for any purpose. And perhaps the language of tomorrow will be more natural with the invention of quantum and biological computers.

**1.1.2 Language Categories**

Programming Languages are often categorized into four categories:

a. Imperative Languages

b. Functional Languages

c. Logical Languages or Declarative Languages

d. Object Oriented Language

The basic architecture of computers has a crucial effect on language design. Most of the popular languages of the past 35 years have been designed around the prevalent computer architecture, called the Von Neumann architecture, after one of its originators, John Von Neumann. These languages are called imperative languages. Programmers using imperative languages must deal with the management of variables and assignment of values to them. The results of this are increased efficiency of execution but laborious construction of programs.

The Functioning Programming paradigm, which is based on mathematical functions, is the design basis for one of the most important non-imperative styles of languages. This style of programming is supported by functional or applicative, programming language. LISP is first functional programming language. The objective of the design of a functional programming language is to mimic mathematical functions to the greatest extent possible. This results in an approach to problem solving that is fundamentally different from methods used with imperative languages. Programming that uses a form of symbolic logic as a programming language is often called logic programming, and languages based on symbolic logic are called programming languages or declarative languages. Prolog is example of logical language.

Procedure-oriented programming, which was the most popular software development paradigm in the 1970’s, focuses on subprograms and sub-program libraries. Data are set to sub-programs for computations. Data-oriented programming focuses on abstract data types. Languages that support data-oriented programming are often called object-oriented-based languages. A language that is object oriented must provide support for three key languages features; abstract data types, inheritance and a particular kind of dynamic binding.

**1.2 HIGH LEVEL PROGRAMMING LANGUAGES**

Advance computer programming language that is not limited by the type of computer or for one specific job and more easily understood. In other words, a programming language designed to suit the requirement of the programmer and independent of the internal machine code of any particular computer, is called high level programming language. For example, BASIC, FORTRAN, C++, Java and Visual Basic are high level programming languages.

In contrast, low-level language, such as assembly languages closely reflect the machine codes of specific computers and, therefore, described as machine-oriented languages. Unlike low-level languages, high-level languages are relatively easy to learn because the instructions bear a closer resemblance to everyday language, and because the programmer does not require a detailed knowledge of the internal working of the computer. Each instruction in a high-level language is equivalent to several machine-code instructions. High-level programs are, therefore, more compact than equivalent low-level programs. However, each high-level instruction must be translated into machine code, by either a compiler or an interpreter, before it can be executed by a computer. High-level languages are designed to be portable, that is programs written in a high-level language can be run on any computer that has a compiler or interpreter for those particular languages.

**1.2.1 Syntax and Semantics of the Languages**

It is very important for any programming languages that it is easily understandable to the programmers and they are able to determine how expressions, statements and program module of the language are formed. Programmers must be able to write the software by the help of manual of programming language.

Syntax of a programming language is the form of its expressions, statements and program modules. And the semantic of a programming language is the meaning given to the various syntactic structure.

For example in C we can take the decision by the following statement:

If (Conditional expression)

{

…………………….

Statements;

…………………….

…………………….

}

The meaning of above syntactical structure is that if conditional expression have true value statements of the blocks will be executed. We have seen that, like any other natural language, computer languages behave as communication media:

a. human to machine

b. machine to machine

c. machine to human

Unlike humans, machines are not fault tolerant, hence programming is often viewed as a “Complex task”. Before switching on the design of compiler, it is very much required to know about the structure of the programs.

**1.3 THE STRUCTURE OF THE PROGRAM**

We can analyze a computer program on 4 levels:

a. lexical structure level

b. syntactic structure level

c. contextual structure level

d. semantic structure level

**1.3.1 Lexical Structure**

The lexical level of any program is lowest level. At this level, computer programs are viewed as simple sequence of lexical items called tokens. In fact at this level, programs are considered as different groups of strings that make sense. Such a group is called a token. A token may be composed of a single character or a sequence of characters. We can classify tokens as being either:

**Identifier** – in a particular language names chosen to represent data items, functions and procedures, etc are called identifiers. Considerations for the identifier may differ from language to language, for example, some computer language may support case sensitivity and others may not. Number of characters in identifiers are also dependent on the design of the computer language.

**Keywords** – are the names chosen by the language designer to represent facts of particular language constructs which cannot be used as identifiers (sometimes referred to as reserved words)

Let us see example of language C

int id ;

here id is identifier

struct idl {

int a ;

char b;

}

Here struct is keyword and idl is identifier

**Operators** – in some languages, special “keywords” used to identify operations to be performed on operands, that is math operators.

For example in language C:

a. Arithmathematical operators (+, -, \*,/,%)

b. Logical operators (>, = >, <, < =, = =, !, ! =)

**Separators** - These are punctuation marks used to group together sequences of tokens that have a “unit” meaning. When computing text, it is often desirable to include punctuation, where these are also used (within the language) as separators, we must precede the punctuation character with what is called an **escape character** (usually a back slash `\’).

**Literals** – Some languages support literals which denote direct values. For example:

a. Numbers that is 1, -123, 3, 14, 6.02

b. Characters, that is `a’

c. String, that is “Some text”

**Comments** – We know that a good program is one that is understandable. We can increase understandability by including meaningful comments into our code. Comments are omitted during processing. The start of a comment is typically indicated by a key word (that is comment) or a separator, and may also be ended by a separator.

**1.3.2 Syntactic Structure**

The syntactic level describes the way that program statements are constructed from tokens. This is always very precisely defined in terms of a context-free grammars. The best known examples are BNF (Backus Naur Form) or EBNF (Extended Backus Naur Form). Syntax may also described using a syntax tree.

Let us see an EBNF example:

<sum> ═> <operands>

<operator>

<operand>

<operand> ═> <number> / (<sum>)

<operator> ═> + / -

The syntax tree for above EBNF will be as follows:

<sum>

<operand1> <operator> <operand2>

<number> <number>

`+’ `-‘

Fig 1.1

**1.3.3 Contextual Structure**

The contextual level of analysis is concerned with the “Context” in which program statements occur. Program statements usually contain identifier whose value is dictated by earlier statements (especially in the case of the imperative or object oriented paradigms). Consequently, the meaning of a statement is dependent on “what has gone before”, that its context. Context also determines whether a statement is “legal” or not (context conditions – a data item must “exist before it can be used”).

For example in language C:

int C = a + b ; } not valid, since a and b are not defined

int a;

int b ; this is valid

int C = a + b;

**1.4 INTRODUCTION OF TRANSLATORS**

Let us consider the translation from French to English. When translating from French to English, the translator must comprehend each French sentence and produce a corresponding sentence in English. Human translator receives a text that is written in the source language (here it is French) and he decomposes it in sentences and words using his knowledge of the language. This “knowledge” is stored in his “internal memory”, that is his brain, and it consists of the vocabulary of the language (used to recognize the words) and the grammar of the language (used to analyze the correctness of the sentences). Once the translator recognizes all the words and understands the meaning of the sentence, he starts the process of translating.

Tips – For any translator, the most difficult part comes from the fact that he or she can not translate word by word because the two language do not have the same sentence pattern’s and in spite of this, the meaning of the sentence in both languages must be the same.

A translator in computer science follows same sequence of operation that is performed by a human translator. Fortunately for the construction of translators, for the computer languages are far simpler than any spoken language, hence the task of writing a program that translates from source language to the target language is far more simple than translating from French to English. So, finally we can define translators as follows:

*“Translators are the software programs, which can translate programs written into source language to target language”*

Source Language Translators Target Language

(Software Programs)

Fig. 1.2

Let us discuss some translators, such as Assemblers, Interpreters and Compilers.

**1.4.1 Assembler**

An Assembler is a translator for an assembly language of a computer. An assembly language is the lowest level programming language for a computer. It is peculiar to a certain computer system and is hence machine-dependent.

Assembly Language Assembler Machine Language

(object code)

Fig. 1.3

**1.4.2 Compilers**

Compilers are system programs that translates an input program in a high level language into its machine language equivalent. Checks for errors. Optimizes the code. Prepares the code for execution. Compilers are the translators, which translate a program written in high-level language (like C++, FORTRAN, COBOL, PASCAL) into machine code for some computer architecture (such as on Intel Pentium architecture). The generated code can be later executed many times against different data each time.

Source program

(High-level language Program) Compiler Object Program Loader or Result

Linkage Editor (or Object Program Output)

Other programs and data

Fig 1.4

In above figure compiler performs of converting a program written on one programming language (that is HLL program) into object code. Now another software processor (which is called loader, also known as linkage editor) performs some very low-level processing of object code in order to convert it into a ready-to-run program in the machine language. Loader is the program from which actually runs on the computer, reading input data, if any, and producing results.

**1.4.3 Interpreter**

An interpreter reads a source program written in a high-level programming language as well as data for this program, and it runs the program against the data to produce some results.

a. Is a systems program that looks at and executes programs on a line-by-line basis rather than producing object code.

b. Whenever the programs have to be executed repeatedly, the source code has to be interpreted every time. In contrast, the compiled programs create an object code and this object code will be executed.

c. Entire source code needs to be present in memory during execution as a result of which the memory space required is more when compared to that of a compiled code.

d. Used in test environments as overhead of compilation is not there.

**1.4.4 Loaders**

Loader is system programs. Loader loads the binary code in the memory ready for execution. Transfer the control to 1st instruction. Loaders are responsible for locating program in the main memory every time it is being executed.

There are various loading schemes:

a. Assemble-and-go loader – the assembler simply places the code into memory and loader executes a single instruction that transfers control to the starting instruction of the assembled program. In this scheme, some portion of the memory is used by the assembler itself which would otherwise have been available for the object program.

b. Absolute Loader – object code must be loaded into the absolute addresses in the memory to run. If there are multiple subroutines, then each absolute address has to be specified explicitly.

c. Relocating Loader – this loader modifies the actual instructions of the program during the process of loading a program so that the effect of the load address is taken into account.

**1.4.5 Linkers**

Linking is the process of combining various pieces of code and data together to form a single executable that can be loaded in memory. Linking is of two main types:

a. Static Linking – All references are resolved during loading at linkage time.

b. Dynamic Linking – References made to the code in the external module are resolved during run time. Takes advantage of the full capabilities of virtual memory. The disadvantage is the considerable overhead and complexity incurred due to postponement of actions till run time.

**1.5 WHY STUDY COMPILERS**

You may never write a commercial compiler, but that’s not why we study compilers. We study compiler construction for the following reasons:

a. Writing a compiler gives a student experience with large scale applications development. Your compiler program may be the largest program you write as a student. Experience working with really big data structure and complex interactions between algorithms will help you out on your next big programming projects.

b. Compiler writing is one of the shining triumphs of Computer Science theory. It demonstrates the value of theory over the impulse to just “hack up” a solution.

c. Compiler writing is a basic element of programming language research. Many language researchers write compilers for the language they design.

d. Many applications have similar properties to one or more phases of a compiler, and compiler expertise and tools can help an application programmer working on other projects beside compilers.

**1.6 WHAT IS THE CHALLENGE IN COMPILER DESIGN?**

**Co**mpiler writing is not an easy job. It is very challenging and you must have lot of knowledge

about various fields of computer science. Let us discuss some of the challenges:

a. Many Variations

i. Many programming languages (FORTRAN, C++, Java)

ii. Many programming paradigms (that is, object-oriented, functional, logical)

iii. Many computer architectures (that is, object-oriented, functional, logical)

iv. Many operating systems (that is Linux, Solaris, Windows)

b. Qualities of a Good Compiler (in order of Importance)

i. The compiler itself must be bug-free

ii. It must generate correct machine code

iii. The generated machine code must run fast

iv. The compiler itself must run fast (Compilation time must be proportional to program size)

v. The compiler must be portable (that is modular, supporting, separate compilation)

vi. It must print good diagnostics and error messages

vii. The generated code must work well with existing debuggers

viii. Must have consistent and predictable optimization

c. Building s Compiler Requires Knowledge of:

i. Programming languages (parameter passing, variable scoping, memory allocation etc.)

ii. Theory of automata, context free languages etc

iii. Algorithms and data structure (hash tables, graph algorithms, dynamic programming etc)

iv. Computer Architecture (Assembly Programming)

v. Software Engineering

d. Instruction Parallelism

Handling of out of order execution and branch prediction is a challenge in compiler design.

e. Parallel Algorithms

Incorporating the concept of grid computing and multi-core computers.

**1.7 COMPILER REQUIREMENTS**

As we shall be implementing several compilers, it is important to understand which requirement compilers should satisfy. We discuss in each case to what extent it is relevant to this course.

i. Correctness – Correctness is absolutely paramount. A buggy compiler is next to useless in practice. Since we cannot formally prove the correctness of your compilers, we use extensive testing. This testing is end-to-end, verifying the correctness of the generated code on sample inputs.

ii. Efficiency – In a production compiler, efficiency of the generated code and also efficiency of the compiler itself are important considerations. The early emphasis on correctness has consequences for your approach to the design of the implementation. Modularity and simplicity of the code are important for two reasons: first, your code is much more likely to be correct, and second, you will be able to respond to changes in the source language specification from lab to lab much more easily.

iii. Interoperability – Programs do not run in isolation, but are linked with library code before they are executed, or will be called as a library from other code. This puts some additional requirements on the compiler, which must respect certain interface specifications. Your generated code will be required to execute correctly in the environments on the lab machines. This means that you will have to respect calling conventions early on (for example, properly save callee-save registers) and data layout conventions later, when your code will be calling library functions.

iv. Usability – A compiler interacts with the programmer primarily when there are errors in the program. As such, it should give helpful error messages. Also, compilers may be instructed to generate debug information together with executable code in order to help uses debug runtime errors in their program.

v. Retargetability – At the outset, we think of a compiler of going from one source language to one target language. In practice, compilers may be required to generate more than one target from a given source (for example, x86-64 and ARM code), sometimes at very different levels of abstraction (for example, x86-64 assembly or LLVM intermediate code).

**1.8 THE COMPONENTS OF A COMPILER**

A compiler is composed of several components also called phases, each performing one specific task. Each of these components performs at a slightly higher language level than the previous. The compiler must take an arbitrarily-formed body of text, and translate it into a binary stream that computer can understand. The complete compilation procedure can be divided into six phases and these phases can be grouped in two parts, as follows:

a. Analysis – In this part source program is broken into constituent pieces and creates an intermediate representation. Analysis can be done in three phases:

i. Lexical Analysis

ii. Syntax Analsyis

iii. Semantic Analysis

b. Synthesis – Synthesis constructs the desired target program from intermediate representation. Synthesis can be done in following three phases:

i. Intermediate code generation

ii. Code optimization

iii. Code generator

Tips – Every phase of compilation can interact with a special data structure called symbol table and with error handler.

Block diagram of Compiler Source Program

Lexical Analyzer

Syntax Analyzer Analysis

Language

Dependent Symbol

Front End Table Semantic Analyzer Error

Manager Handler

Intermediate Code Generator

Target Code Optimizer

Dependent Synthesis

Back End Code Generator

Target Program

Fig. 1.5: Phases of a Compiler

Sometimes we divide synthesis as follows:

a. Intermediate code generation

b. Code optimization

c. Storage allocation

d. Code generation

Here, we are looking at storage allocation as a different phase, but in previous assumption it was the part of code generation.

Now block diagram can be drawn as follows:

|  |  |  |
| --- | --- | --- |
| Source Program | | |
| Symbol Table Manager | Local Analyzer | Error Handler |
|  |
| Syntax Analyzer |
|  |
| Semantic Analyzer |
|  |
| Intermediate Generator |
|  |
| Storage Allocation |
|  |
| Code Optimizer |
|  |
| Code Generator |
| Output Target Program | | |

Fig. 1.6: Block Diagram of a Compiler

**1.8.1 Modern Approach of Compiler**

In the past, compilers were divided into many passes to save space. When each pass is finished, the compiler can free the space needed during the pass. Many modern compilers share a common `two stage’ design. The first stage, the `compiler front end’ translates the source language into an intermediate representation. The second stage, the `compiler back end’ works with the internal representation to produce code in the output language.

**a. Compiler Front End**

The compiler front end consists of multiple phases itself, each informed by formal language theory:

i. **Lexical Analysis** – breaking the source code text into small pieces (`tokens’ or terminals’), each representing a single atomic unit of the language, for typically a regular language, so a finite state automation constructed from a regular expression can be used to recognize it. This phase is also called lexing or scanning.

ii. **Syntax Analysis** – identifying syntactic structures of source code. It only focuses on the structure. In other words, it identifies the order of tokens and understand hierarchical structures in code. This phase is also called parsing.

iii. **Semantic Analysis** – is to recognize the meaning of program code and start to prepare for output. In that phrase, type checking is done and most of compiler errors show up.

iv. **Intermediate Language Generation** – an equivalent to the original program is created in an intermediate language.

WHERE IS THE FIGURE HERE? PLEASE FIX IT ACCORDINGLY.

Fig. 1.7 – Structure of a Typical Compiler

b. **Compiler Back End**

While there are applications where only the compiler front end is necessary, such as static language verification tools, a real compiler hands the intermediate representation generated by the front end to the back end, which produces a functional equivalent program in the output language. This is done in multiple steps as follows:

i. Optimization – the intermediate language representation is transformed into functionally equivalent but faster (or smaller) forms.

ii. Code generation – the transformed intermediate language is translated into the output language, usually the native machine language of the system. This involves resource and storage decisions, such as deciding which variables to fit into registers and memory and the selection and scheduling of appropriate machine instructions.

**1.8.2 Lexical Analyzer (Scanner)**

This module has the task of separating the continuous string of characters into distinctive groups that make sense. Such a group is called a token. A token may be composed of a single character or a sequence of characters (this sequence of characters is called lexeme).

In order to perform this task, the lexical analyzer must know the key words, identifiers, operators, delimiters and punctuation symbols of the language to be implemented. At the same time as forming characters into symbols the lexical analyzer should deal with (multiple) spaces and remove comments and any other characters not relevant to the later stages of analysis. We can say that lexical analyzer design must:

a. Specify the token of the language

b. Efficiently recognize the tokens

To specify the token of the language, regular expression concept from automata theory is used and recognition of token is done by deterministic finite automata (we shall see in text chapter).

Now let us do the analysis of the following:

Sum : = Old sum – Value/100

|  |  |
| --- | --- |
| Lexeme (collection of characters) | Token (category of lexeme) |
| Sum  :=  Old sum  -  Value  /  100 | Identifier  assignment operator  identifier  subtraction operator  identifier  division operator  integer constant |

Fig. 1. 8

Let us see another example:

Position := Initial + rate + 70

This statement is grouped into 7 tokens as follows:

|  |  |
| --- | --- |
| Lexeme | Token (category of lexeme) |
| Position  :=  initial  +  rate  \*  70 | Identifier  assignment operator  identifier  addition operator  identifier  multiplication operator  integer constant |

Fig. 1. 9

After recognizing the token lexical analyzer passes them to syntax analyzer. Another way of looking at it is that lexical analyzer usually has no context to work with. As it is processing one symbol it has no knowledge of any of the symbols that preceded or will follow this symbol. There are widely available tools that can be used for producing a lexical analyzer for a language (we shall see some of them later).

**1.8.3 Syntax Analyzer (Parser)**

Syntax analyzer is the module in which the overall structure is identified and involves an understanding of the order in which the symbols in a program may appear. We know that it was the scanner’s duty to recognize individual words or tokens of the language. The lexical analyzer does not, however, recognize if these words have been used correctly. The main task of parser is to group the tokens into sentences, that is, to determine if the sequence of tokens that have been extracted by the syntax analyzer are in the correct order or not. In other words, until the syntax analyzer (parser) is reached the tokens have been collected with no regard to the whole context of the program as whole. The parser analyzes the context of each token and groups the token in declarations, statements and control statement.

Tips – In the process of analyzing each sentence, the parser builds abstract tree structures.

A tree is a useful representation of a program segment because it facilitates transformations of the program that may lead to possible minimization of the machine instructions that are needed to carry out the required operations, that is, optimization. Let us see the procedure by an example:

Sum : = Old sum – Value/100

There are 7 tokens, sum (identifier) `:=’ (assignment operator), old sum (identifier), `-‘ (subtraction operator), value (identifier), / (division operator) and 100 (integer constant) as output for lexical analysis. Now these tokens work as input to the parser.

Parser will produce the following syntax tree:

Assign op (:=)

Sum sub op(-)

Old sum divide op (/)

Value 100

Fig. 1.10 – Syntax Tree as Output of Parser

So we can say that syntax analysis is not only the key phase of the analysis stage of compilation but provides the framework for the compiler as a whole. It drives the lexical analysis phase and builds the structure upon which semantic analysis is performed. Syntax analysis also provides a framework for a whole range of source code analysis tools, including measuring tools of various sorts, cross-referencers, layout tools and so on.

**1.8.4 Semantic Analyzer**

Some features of programming cannot be checked in a single left to right scan of the source code without setting up arbitrary sized tables, to provide access to information that may be an arbitrary distance away, when it required. For example, information concerning the type and scope of variables falls into this category. The semantic analyzer gathers type information and checks the tree produced by the syntax analyzer for semantic errors. Let us see the statement; and consider rate is real. Sum : = Old sum + Rate & 60 ; Here in this statement the semantic analyzer might add a type conversion node, say, inttoreal, to the syntax tree to convert the integer to real quantity.

Output of syntax analysis:

assign op

sum add op

old sum mul op

rate inttoreal

num (60)

Fig. 1.12

**1.8.5 Symbol Table Manager**

Though the parser determines the correct usage of tokens, and whether or not they appeared in the correct order, it still did not determine whether or not the program said anything that made sense. This is type of checking that occurs at the semantic level. In order to perform this task, the compiler makes use of a detailed system of lists, known as the symbol tables. The compiler needs a symbol table to record each identifier and collect information about it.

Tips – For a variable the symbol table might record its type-expression (integer, float etc), its scope (where the variable can be used), and its location in run-time storages, etc. For a procedure the symbol table might record the type-expression of its arguments and the type expression of its returned value.

The symbol table manager has a FIND function that returns a pointer to the descriptor for an identifier (descriptor is a record which contains all the information about identifier) when given its lexeme. Compiler phases use this pointer to read and/or modify information about the identifier. FIND returns a NULL pointer if there is no record for a given lexeme. The INSERT function in the symbol table manager inserts a new record into the symbol table when given its lexeme.

**1.8.6 Intermediate Code Generator**

After semantic analysis many compilers generated an Intermediate representation of the source program that is both easy to produce and easy to translate into the target program. There are a variety of forms used for intermediate code. One such form is called three-address code and looks like code for a memory-memory machine where every operation reads operands from memory and writes results into memory. The intermediate code generator usually has to create temporary location to hold intermediate results. For example see following statement

Sum : = Old sum + Rate & 60 ;

After the compiler broke this stream of text down into tokens, parsed the tokens and created the tree and finally performed the necessary type checking, it might creates a sequence of three-address instructions. Three address instructions contain no more than, three operands (registers) per instruction and in addition to an assignment, contain only one other operator. Three-address instructions also assume on unlimited supply of registers.

temp1 : = inttoreal (60)

temp2 : = rate & temp1

temp3 : = old sum + temp2

sum : = temp3

where temp1, temp2 and temp3 are the names of three temporary locations created by the intermediate code generator.

**1.8.7 Code Optimization**

The structure of the tree that is generated by the parser can be rearranged to suit the needs of the machine architecture or the tree can be restructured in order to produce an object code that run faster. What results is a simplified tree that may be used to generate the object code. An example of such an optimization is the following FOR Loop:

for (int i = 0; < 2000 ; i + +)

{

x : = y + 3 ;

q : = a \* i ;

}

In this loop there are instructions that are not related at all to the variables `a’ of the for statement, such as x: = y + 3. In the above program, this instruction is executed unnecessarily 2000 times. The object code can be generated such that the machine code for that line is moved outside the for’s body, (either before or after the for command). The resulting object code is more efficient at run time because the loop has been significantly tightened and the above mentioned instruction is executed only once, instead of 2000 times. This is called loop optimization. In intermediate code generation phase, we have seen three-address instructions as follows:

temp1 : = inttoreal (60)

temp2 : = rate & temp1

temp3 : = old sum + temp2

sum : = temp3

We can optimize the code, by applying abstract algebraic rules to it. A subsequent optimization of the above code might be:

temp1 : = rate & inttoreal (60)

sum : = Old sum + temp1

This is sometimes called local optimization. Once the intermediate code has been optimized, the compiler can generate the final code in its machine-readable form.

**1.8.8 Storage-allocation**

Every constant and variable appearing in the program must have storage space allocated for its value during the storage allocation phase. This storage space may be one of the following types:

a. Static Storage – If the life time of the variable is life time of the program and the space for its value once allocated cannot later be released. This kind of allocation is called static storage allocation.

b. Dynamic Storage – We follow dynamic storage allocation if the life time is a particular block or function or procedure in which it is allocated so that it may be released when the block of function or procedure in which it is allocated is left.

c. Global Storage – If its life time is unknown at compile time and it has to be allocated and de-allocated at run time. The efficient control of such storage usually implies run-time overheads.

Tips – After space is allocated by the storage allocation phase, an address, containing as much as is known about its location at compile time, is passed to the code generator for its use.

**1.8.9 Code Generation**

This is the part of the compiler where native machine code is actually generated. If the target machine has registers the target program. For example, for statement

Sum : = old sum + rate & 60

Now we can understand that sum, old sum and rate are identifier, let us say them id1, id2, and id3.

That is: id1 : = id2 + id3 \* 60

MOVF id3, R2,

MULF # 60.0, R2,

MOVF id2, R1

ADDF R2, R1

MOVF R1,id1,

During this step, the compiler has to map address names from the three-address intermediate code onto the very finite amount of registers had by the machine. The final result is a piece of code that is (mostly) executable. None of the remaining phases prior the actual execution of the program are a part of the compiler’s responsibility. What remains to be done is to link the program and build a binary executable. In this phases, various libraries as well as the main executable portion of the program are combined to form a fully-fledged application. There are various types of linking, most of which are done prior to run time. The final phase that a program must pass through prior to execution is for it to be loaded into memory. In this phase, the addresses in the binary code are translated from logical addresses, and any final run time binding are performed.

**1.8.10 Peephole Optimization**

The object code can be further optimized by a peephole optimizer. This process is present in a more sophisticated compiler. The purpose is to produce a more efficient object program that can be executed in a shorter time. Optimization is performed at a local level and it takes advantage of certain operator properties such as cumulatively, associativity and distributivity. To see a summary of the entire process of compilation from start to finish, refer to following figure.

**1.6 Intermediate Code Generation**

On a logical level, the output of the syntax analyzer is some representation of a parse tree. The intermediate code generation phase transforms this parse tree into an intermediate-language representation of the source program.

**Three-Address Code**

One popular type of intermediate language is what is called “three-address code”. A typical three-address code statement is

A : == B op C

where A, B, and C are operands and op is a binary operator

The parse tree in Fig. 1.4(a) might be converted into the three-address code sequence

expression

expression expression

expression expression

A / B \* C

(a)

expression

expression expression

expression expression

A / B \* C

Fig. 1.4 – Parse Trees

T1 :== A/B

T2 :== T1 \* C

where T1 and T2 are names of temporary variables

In addition to statements that use arithmetic operators, an intermediate language needs unconditional and simple conditional branching statements, in which at most one relation is tested to determine whether or not a branch is to be made. Higher-level flow of control statements such as while-do statements, or if-then-else statements are translated into these lower-level conditional three-address statements.

statement

if-statement

if ( conditional ) non-if-statement

relation goto statement

expression relational-op expression goto label

const eq id

Fig. 1.5: Parse of if-statement (1.2)

statement

while-statement

while condition do statement

condition & condition assignment

relation relation location ← exp

id(A) exp + exp

exp relop exp exp relop exp

id(A) id(B)

id(A) > id(B) id(A) ≤ exp - exp

exp \* exp const(5)

const(2) id(B)

Fig. 1.6 – Parse Tree for While-statement

L1: if A > B goto L2

goto L3

L2: T1 := 2 \* B

T2 := T1 – 5

if A ≤ T2 goto L4

goto L3

L4: A := A + B

goto L1

L3:

Fig. 1.7 – Intermediate Code for While-statement

Example 1.3. Consider the following while-statement

while A > B & A < ==2\*B-5 do

A :== A + B;

which has the corresponding token stream

while [id, n1] > [id, n2] & [id, n1] ≤ [const, n3] \* [id, n2]

- [const, n4] do [id, n1] ← [id, n1] + [id, n2];

Here n1, n2, n3, and n4 stand for pointers to the symbol table entries for A, B, 2 and 5 respectively. The parse tree for this statement might plausibly be the one shown in Fig 1.6. We use “exp” for “expression”, “relop” for “relational operator” and we indicate parenthetically the particular name or constant to which each instance of token **id** and **const** refer. The actual algorithms by which parse trees such as Fig. 1.6 can be translated to intermediate code will be discussed later. However, we can now show what the intermediate code should look like. A straightforward algorithm for translation would produce intermediate code like that shown in Fig. 1.7. The jumps over jumps, such as in the first two statements, can be cleaned up during the code-optimization or code generation phase. Later we will talk about intermediate-code generation by syntax-directed translation, a technique in which the actions of the syntax analysis phase guide the translation. These shows how to define intermediate-language constructs in terms of the syntactic constructs found in a programming language. They also show how the intermediate code can be generated as syntax analysis takes place.

**1.7 Optimization**

Object programs that are frequently executed should be fast and small. Certain compilers have within them a phase that tries to apply transformations to the output of the intermediate code generator, in an attempt to produce an intermediate-language version of the source program from which a faster or smaller object-language program can ultimately be produced. This phase is popularly called the optimization phase.

The term “optimization” in this context is a complete misnomer, since there is no algorithmic way of producing a target language program that is the best possible under any reasonable definition of “best”. Optimizing compilers merely attempt to produce a better target program than would be produced with no “optimization”. A good optimizing compiler can improve the target program by perhaps a factor of two in overall speed, in comparison with a compiler that generates code carefully but without using the specialized techniques generally referred to as code optimization.

**Local Optimization**

There are “local” transformations that can be applied to a program to attempt an improvement. For example, in Fig. 1.7 we saw two instances of jumps over jumps in the intermediate code, such as

if A > B goto L2

goto L3

L2: (1.3)

This sequence could be replaced by the single statement

if A ≤ B goto L3 (1.4)

Sequence (1.3) would typically be replaced in the object program by machine statements which:

a. compare A and B to set the condition codes

b. jump to L2 if the code for > is set, and

c. jump to L3

Sequence (1.4), on the other hand, would be translated to machine instructions which:

d. compare A and B to set the condition codes and

e. jump to L3 if the code for < or == is set

If we assume A > B is true half the time, then for (1.3) we execute (1) and (2) all the time and (3) half the time, for an average of 2.5 instructions. For (1.4) we always execute two instructions, a 20% savings. Also, (1.4) provides a 33% space saving if we crudely assume that all instructions require the same space.

Another important local optimization is the elimination of common sub-expressions. Provided A is not an alias for B or C, the assignments

A := B + C + D

E := B + C + F

might be evaluated as

T1 := B + C

A := T1 + D

E := T1 + F

Taking advantage of the common subexpression B + C. Common subexpressions written explicitly by the programmer are relatively rare, however, A more productive source of common subexpressions arises from computations generated by the compiler itself. Chief among these is subscript calculation. For example, the assignment

A [1] := B[1] + C[1]

will, if the machine memory is addressed by bytes and there are, say, four bytes per word, require 4\*I to be computed three times. An optimizing compiler can modify the intermediate program so that the calculation of 4\*I is done only once. Note that it is impossible for the programmer to specify that this calculation of 4\*I be done only once in the source program, since these address calculations are not explicit at the source level.

**Loop Optimization**

Another important source of optimization concerns speedups of loops. Loops are especially good target for optimization because programs spend most of their time in inner loops. A typical loop improvement is to move a computation that produces the same result each time around the loop to a point in the program just before the loop is entered. Then this computation is done only once each time the loop is entered. Then this computation is done only once each time the loop is entered, rather than once for each iteration of the loop. Such a computation is called loop invariant.

**1.8 Code Generation**

The code-generation phase converts the intermediate code into a sequence of machine instructions. A simple-minded code generator might map the statement A: =B+C into the machine code sequence.

LOAD B

ADD C

STORE A

However, such a straightforward macro-like expansion of intermediate code into machine code usually produces a target program that contains many redundant loads and stores and that utilizes the resources of the target machine inefficiently. To avoid these redundant loads and stores, a code generator might keep track of the run-time contents of registers. Knowing what quantities reside in registers, the code generator can generate loads and stores only when necessary.

Many computers have only a few high-speed registers in which computations can be performed particularly quickly. A good code generator would therefore attempt to utilize these registers as efficiently as possible. This aspect of code generation, called register allocation, is particularly difficult to do optimally, but some heuristic approaches can give reasonably good results.

**1.9 Bookkeeping**

A compiler needs to collect information about all the data objects that appear in the source program. For example, a compiler needs to know whether a variable represents an integer or a real number, what size an array has, how many arguments a function expects, and so forth. The information about data objects may be explicit, as in declarations, or implicit, as in the first letter of an identifier or in the context in which an identifier is used. For example, in FORTRAN, A(I) is a function call if A has not been declared to be an array.

The information about data objects is collected by the early phases of the compiler – lexical and syntactic analysis – and entered into the symbol table. For example, when a lexical analyzer sees an identifier MAX, say, it may enter the name MAX into the symbol table if it is not already there, and produce as output a token whose value component is an index to this entry of the symbol table. if the syntax analyzer recognizes a declaration integer MAX, the action of the syntax analyzer will be to note in the symbol table that MAX has type “integer”. No intermediate code is generated for this statement.

The information collected about the data objects has a number of uses. For example, if we have the expression A + B, where A is of type integer and B of type real, and if the language permits an integer to be added to a real, then on most computers code must be generated to convert A from type integer to type real before the addition can take place. The addition must be done in floating point, and the result is real. If mixed-mode expressions of this nature are forbidden by the language, then the compiler must issue an error message when it attempts to generate code for this construct.

The term semantic analysis is applied to the determination of the type of intermediate results, the check that arguments are of types that are legal for an application of an operator, and the determination of the operation denoted by the operator (e.g. + could denote fixed or floating add, perhaps logical “or”, and possibly other operations as well). Semantic analysis can be done during the syntax analysis phase, the intermediate code generation phase, or the final code generation phase.

**1.10 Error Handling**

One of the most important functions of a compiler is the detection and reporting of errors in the source program. The error messages should allow the programmer to determine exactly where the errors have occurred. Errors can be encountered by virtually all of the phases of a compiler. For example:

a. The lexical analyzer may be unable to proceed because the next token in the source program is misspelled.

b. The syntax analyzer may be unable to infer a structure for its input because a syntactic error such as a missing parenthesis has occurred.

c. The intermediate code generator may detect an operator whose operands have incompatible types.

d. The code optimizer, doing control flow analysis, may detect that certain statements can never be reached.

e. The code generator may find a compiler-created constant that is too large to fit in a word of the target machine.

f. While entering information into the symbol table, the bookkeeping routine may discover an identifier that has been multiply declared with contradictory attributes.

Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic message. Once the error has been noted, the compiler must modify the input to the phase detecting the error, so that the latter can continue processing its input, looking for subsequent errors. Good error handling is difficult because certain errors can mask subsequent errors. Other errors, if not properly handled, can spawn an avalanche of spurious errors. Techniques for error recovery are discussed in the future..

**1.11 Compiler-Writing Tools**

A number of tools have been developed specifically to help construct compilers. These tools range from scanner and parser generators to complex systems, variously called compiler-compilers, compiler-generators or translator-writing systems, which produce a compiler from some form of specification of a source language and target machine. The input specification for these systems may contain:

a. a description of the lexical and syntactic structure of the source languageb. a description of what output is to be generated for each source language construct

c. a description of the target machine

In many cases, the specification is merely a collection of programs fitted together into a framework by the compiler-compiler. Some compiler-compilers, however, permit a portion of the specification of a language to be nonprocedural rather then procedural. For example, instead of writing a program to perform syntax analysis, the user writes a context-free grammar and the compiler-compiler automatically converts that grammar into a program for syntax analysis.

While a number of useful compiler-compilers exist, they have limitations. The chief problem is that there is a tradeoff between how much work the compiler-compiler can do automatically for its user and how flexible the system can be. For example, it is tempting to assume that lexical analyzers for all languages are really the same, except for the particular keywords and signs recognized. Many compiler-compilers do in fact produce fixed lexical analysis routines for use in the generated compiler. These routines differ only in the list of keywords recognized, and this list is supplied by the user. This approach is quite valid, but may be unworkable if it is required to recognize nonstandard tokens such as identifiers that may include characters other than letters and digits. More general approaches to the automatic generation of lexical analyzers exist, such as those described, but these require the user to supply more input to the compiler-compiler, i.e. to do more work. The principal aids provided by existing compiler-compilers are:

a. Scanner generators – The “built-in” approach described above and regular expression based techniques described are the most common approaches.

b. Parser generators – Almost every compiler-compiler provides one. The reason is twofold. First, while parsing represents only a small part of compiler construction, having a fixed framework in which parsing is done can be a great aid in the organization of the entire compiler. Second, the parsing phase is unique among the compiler phases in that a notation exists – the context-free grammar described – which is sufficiently nonprocedural to reduce the work of the compiler writer significantly, sufficiently general to be of use in any compiler, and sufficiently developed to permit efficient implementations to be generated automatically. One significant advantage of using a parser generator is increased reliability. A mechanically-generated parser is more likely to be correct than one produced by hand.

c. Facilities for code generation – Often a high-level language especially suitable for specifying the generation of intermediate, assembly, or object code is provided by the compiler-compiler. The user writes routines in this language and, in the resulting compiler, the routines are called at the correct times by the automatically generated parser. A common feature of compiler-compilers is a mechanism for specifying decision tables that select the object code. These tables become part of the generated compiler, along with an interpreter for these tables, supplied by the compiler-compiler. The bibliographic notes contain references to a number of compiler-compiler systems.

**1.12 Getting Started**

Although a compiler is just another program in a system, it usually is an important program that will be used by many people. Therefore, before writing any code a would-be compiler designer should give some thought to the following issues. A new compiler may be for a new source language, or produce new object code or both. If the source language is new, how will it be used? Should compilation speed be of greater importance than the quality of output code? How important are good error diagnostics and good error recovery?

The nature of the target machine and operating environment likewise should be considered, for they have a strong influence on what the compiler will look like and what code generation strategies it should use. In terms of writing the compiler, the implementation language, the programming environment, and the available tools are most important, for they determine how quickly the compiler can be built. For example, it is painfully obvious that systems programming languages, such as BLISS, C and PASCAL, are much better-suited to compiler construction than are APL, COBOL or FORTRAN.

Compiler-writing, like any other large software effort, is an exercise in compromise. The design, therefore, should be such that change and modification can be readily accommodated throughout the birth and life of the compiler. The use of some of the compiler-building tools we discuss can be a significant help in this direction.

**Bootstrapping**

A compiler is characterized by three languages: its source language, its object language and the language in which it is written. These languages may all be quite different. For example, a compiler may run on one machine and produce object code for another machine. Such a compiler is often called a cross-compiler. Many minicomputer and microprocessor compilers are implemented this way; they run on a bigger machine and produce object code for the smaller machine. Sometimes we hear of a compiler being implemented in its own languages. This naturally raises the question, “How was the first compiler compiled”? This question may sound like “Who was the firs parent”? but it is not nearly as hard.

Suppose we have a new language L, which we want to make available on several machines, say A and B. As a first step, we might write for machine A a small compiler CSAA† that translates a subset S of language L into the machine or assembly code of A. This compiler can first be written in a language that is already available on A (the assembly language of A if need be).

We then write a compiler CLAS in the simple language S. This program, when run through CSAA, becomes CLAA, the compiler for the complete language L, running on machine A, and producing object code for A. The process is shown in Fig. 1.8.

CLAS → CSAA → CLAA

Fig. 1.8 – Bootstrapping a Compiler

Now suppose we want to produce another compiler for L to run on machine B and to produce code for B. If CLAS has been designed carefully and machine B is not that different from machine A, it should be far less work to convert CLAS into a compiler CLBL which produces object code for B than it is to write a new compiler from scratch. Note that we can now use the full language L to implement CLBL.

† We use the notation CXYZ to stand for language X, written in language Z, and producing object code in language Y. We use A and B to stand for the machine codes of computers A and B.

CLBL → CLAA → CLBA

CLBL → CLBA → CLBB

Fig. 1.9 – Bootstrapping a Compiler to a Second Machine

Using CLBL to produce CLBB, a compiler for L on B, is now a two-step process, as shown in Fig. 1.9. We first run CLBL through CLAA to produce CLBA, a cross-compiler for L which runs on machine A but produces cod for machine B. Then we run CLBL through this cross-compiler to produce the desired compiler for L that runs on machine B and produces object code for B.

**1.8.11 Error Handler**

Detection and reporting of errors in source program is main function of compiler. Error may occur at any phase of compilation. A good compiler must determine the line number of program exactly, where the errors have occurred.

These are the various errors which may occur at different levels of compilation:

a. **The first of these are lexical (scanner) errors.** Some of the most common types here consist of illegal or unrecognized characters, mainly, caused by typing errors. A common way for this to happen is for the programmer to type a character that is illegal in any instance in the language and is never used. Finally, the quite commonly, another type of error that the scanner may detect is an unterminated character or string constant. This happens whenever the programmer types something in quotes and forget the trailing quote. Again, these are mostly typing errors.

b. **The second class of error is syntactic in nature** and is caught by the parser. These errors are among the most common. The really difficult part is to decide from where to continue the syntactical analysis after an error has been found. What happens if the parser is not carefully written or if the error detection and recovery scheme is sloppy, the parser will hit one error and `mess up’ after that, and cascade spurious error messages all throughout the rest of the program. In the case of an error, what one would like to see happening, is to have the compiler skip any improper tokens, and continue to detect errors without generating error messages that are not really an error but a consequence of the first error. This aspect is so important that some compilers are categorized base on how good their error detection system is.

c. **The third type of error is semantic in nature.** The semantic that are used in computer languages are by far simpler than the semantics that are used in spoken languages. This is the case because in computer languages everything is very much defined, there are nonsense implied or used. The semantic errors that may occur in a program are related to the fact that some statements may be correct from the syntactical point of view, but they make no sense and there is no code that can be generated to carry out the meaning of the statement.

d. **The fourth type of error may encounter during code optimization,** for example in control flow analysis, there may exist some statements, which can never be reached.

e. **The fifth type of error may occur during code generation, since in code generation architecture of computer also play an important role.** For example, there may exist a constant that is too large to fit in a word of the target machine.

f. **The sixth type of error may encounter when compiler try to make symbol table entries,** for example there may exist an identifier that has been multiple declaration with contradictory attributes.

**1.9 THE NUMBER OF COMPILER PASSES**

We have seen the structure of compiler. The structure of the compiler is divided into six phases. Each phase have its own unique job to perform. The output of one phase is used as input to another phase. We have already seen that phases can be viewed as analysis phase and synthesis phase. On the basis of regrouping of phase compilers may multi-pass or one pass compilers.

a. A pass is a complete traversal of the source program or a complete traversal of some inter representation of the source program.

b. A pass can correspond to a “phase” but it does not have to!

c. Sometimes a single “pass” corresponds to several phases that are interleaved in time.

d. What and how many passes a compiler does over the source program is an important decision.

**1.9.1 Multi-pass Compilers**

We have already studied a compiler that scan the input source once, produces a first modified form, then scans the first-modified form and produce a second-modified form and so on, until the object form is produced. **Such a compiler is called a multipass compiler. In the case of the multi-pass compiler each function of the compiler can be performed by one pass of the compiler.** For instance, the first pass can read the input source, scan and extract the tokens and store the result in an output file. The second pass can read the file which was produced in the first pass, do the syntactical analysis by building a syntactical tree and associate all the information relating to each node of the tree. The output of the second pass, then is a file containing he syntactical tree. The third pass, can read the output file produced by the second pass and perform the optimization by restructuring the tree structure.

A multi-pass compiler makes several passes over the program. The output of a preceding phase is stored in a data structure and used by subsequent phases.

**Dependency diagram of a typical Multi Pass Compiler:**

Compiler Driver

Calls calls calls

Syntactic Analyzer Contextual Analyzer Code Generator

Input output input output input output

Source text AST Decorated AST Object Code

Fig. 1.14

**1.9.2 One-Pass Compilers**

In a one-pass compiler, when a line source is processed, it is scanned and the tokens are extracted. Then the syntax of the line is analyzed and the tree structure and some tables containing information about each token are built. Finally, after the semantical part is checked for correctness, the code is generated. The same process is repeated for each line of code until the entire program is compiled. Usually, the entire compiler is built around the parser, which will call procedures that will perform different functions.

A single pass compiler makes a single pass over the source text, parsing, analyzing and generating code all at once.

**Dependency diagram of a typical single pass compiler:**

Compiler Driver

Calls

Syntactic Analyzer

Calls Calls

Contextual Code

Analyzer Generator

Fig. 1.15

**1.9.3 Advantages and Disadvantages for Both Single and Multi-pass Compilers**

a. A one pass compiler is fast, since all the compiler code is loaded in the memory at once. It can process the source text without the overhead of the operating system having to shut down one process and start another. Also, the output of each pass of the multi-pass compiler is stored on disk, and must be read in each time the next pass starts.

b. On the other hand, a one pass tends to impose some restrictions upon the program: constants, types, variables and procedures must be defined before they are used. A multi-pass compiler does not impose this type of restrictions upon the user. Operation that cannot be performed because of the lack of information can be deferred to the next pass of the compiler when the text has been traversed and the needed information made available.

c. The components of a one-pass compiler are inter-related much closer than the components of a multi pass compiler. This requires all the programmers working on the project to have knowledge about the entire project. A multi-pass compiler can be decomposed into passes that can be relatively independent, hence a team of programmers can work on the project with little interaction among them. Each pass of the compiler can be regarded as a mini-compiler, having an input source written in one intermediary language and producing an output written in another intermediary language.

**1.9.4 Difference Between Single Pass Compiler and Multi-Pass Compiler**

a. A **one-pass compiler** is a compiler that passes through the source code of each compilation unit only once. A **multi-pass compiler** is a type of compiler that processes the source code or abstract syntax tree of a program several times.

b. A one-pass compilers is faster than multi-pass compilers.

c. A one-pass compiler has limited scope of passes but multi-pass compiler has wide scope of passes.

d. Multi-pass compilers are sometimes called **wide compilers** where as one-pass compiler are sometimes called **narrow compiler**.

e. Many programming languages cannot be represented with single pass compilers, for example **Pascal** can be implemented with a single pass compiler where as languages like **Java** require a multi-pass compiler.

**1.10** **INTRODUCTION TO BOOTSCTRAPPING (COMPILERS)**

When writing a compiler, one will usually prefer to write it in high-level language. A possible choice is to use a language that is already available on the machine where the compiler should eventually run. It is, however, quite common to be in the following situation:

You have a completely new processor for which no compilers exist yet. Nevertheless, you want to have a compiler that not only target this processor, but also runs on it. In other words, you want to write a compiler for a language A, targeting language B (the machine language) and written language B.

The most obvious approach is to write the compiler in language B, But if B is machine language, it is horrible job to write any non-trivial compiler in this language. Instead, it is customary to use a process called “bootstrapping”, referring to the seemingly impossible task of pulling oneself up by the bootstraps. The idea of bootstrapping is simple: You write your compiler in language A (but still let it target B) and then let it compile itself. The result is a compiler from A to B written in B. It may sound a bit paradoxical to let the compiler itself. In order to use the compiler to compile a program, we must already have compiled it, and to do this, we must use the compiler. In a way, it is a bit like the chicken-and-egg paradox. We shall shortly see how this apparent paradox is resolved, but first we will introduce some useful notation.

**1.10.1 Notations**

We will use a notation designed by H. Bratman. The notation is hence called “Bratman diagrams” or, because of their shape, “T-diagrams”. These are some notations:

a. A compiler written in language H that translates language S into language T.

S T

H

b. T-Diagram can be combined in two basic ways

**Now let us see the first T-diagram combination as follows:**

A B B C A C

H H H

c. Two compilers run on the same machine H

* First from A to B
* Second from B to C
* Result from A to C on H

**Now let us see the Second T-diagram Combination as follows:**

A B A K

H H K B

M

a. Translate implementation language of a compiler from H to K

b. Use another compiler from H to K

**Now let us study the following Scenario-1 as follows**:

A H A K

B B H H

H

a. Translate a compiler from A to H written in B

* Use an existing compiler for language B on machine H

**Now let us study the following Scenario-2 as follows:**

A H A H

B B K K

K

b. Use an existing compiler for language B on different machine K

* Result in a cross compiler

**1.10.2 Bootstrapping (Compiling Compilers)**

The basic idea in bootstrapping is to use compilers to compile themselves or other compilers. We do, however, need a solid foundation in form of a machine to run the compilers on.

**Process of Bootstrapping**

a. Write a compiler in the same language

S T

S

b. No compiler for source language yet.

c. Porting to a new host machine.

**First Step in Bootstrap**

A H A H

A A H H

H

a. “quick and dirty” compiler written in machine language H

b. Compiler written in its own language A

c. Result in running but inefficient compiler

**The Second Step in Bootstrap**

A H A H

A A H H

H

a. Running but inefficient compiler

b. Compiler written in its own language A

c. Result in final version of the compiler

The above bootstrapping process relies on an existing compiler for the desired language, albeit running on a different machine. It is, hence, often called “half bootstrapping’. When no existing compiler is available, e.g., when a new language has been designed, we need to use a more complicated process called “full bootstrapping”.

**1.10.3 Porting**

In computer science, porting is the process of adapting software so that an executable program can be created for a computing environment that is different from the one for which it was originally designed (e.g. different CPU, operating system, or third party library). The term’s also used for when software, relative to its implementation cost, the more portable it is said to be.

**The step 1 in Porting**

A K A K

A A H H

H

a. Original compiler

b. Compiler source code retargeted to K

c. Result in Cross Compiler

**The step 2 in Porting**

A K A K

A A K K

H

a. Cross compiler

b. Compiler source code retargeted to K

c. Result in Retargeted Compiler

**1.11 THE TARGET ARCHITECTURE OF COMPILER**

A compiler can generate code either for a real machine or for a virtual machine (VM). A real machine is a piece of hardware equipment, like a microprocessor, capable of understanding an assembly code and executing it. **A virtual machine is a software program that acts like a real machine**. There are advantages and disadvantages, which result from using each one of these machines.

**Real Machine Architecture**. Real machine has the advantage of working very fast: as fast as the hardware of the micro processor works. On the other hand, it has the disadvantage of being just as complicated. Often, generating and formatting instructions for these machines are very technical, tedious and time-consuming. In addition, the programmer has to become familiar with several very technical features of the operating system.

In short, generating code for a CPU is not an easy operation because there are hundreds of codes, and for each code there are a lot of details at the bit and byte level that must be considered. On the other hand, the instructions may be very tiny, and the resulting code that is generated can be very long, making the compiler more complicated. Overall, the instructional process can be very painful.

**1.11.1 Virtual Machine Architecture**

In case of a virtual machine, instructions are different than those of a real machine. In software, things like registers do not necessarily speed up the execution process. Also, instructions must necessarily be simple, since the overhead of decoding each instruction prior to execution becomes quite significant. On one hand, the higher the level of instructions, the less code the compiler has to generate. On the other hand, the simpler the instructions are, the quicker they can be executed by the VM. Fortunately, the demands of two are not mutually exclusive. Instruction must be both high-level, and quick for the software to decode. In addition to this, an extra benefit can be added, that the virtual machine instructions can be designed in accordance with the needs of the compiler. This can really simplify the process of generating object code. The code generated for virtual machine best fit the needs of the compiler, could be measurable shorter than code generated for a general-purpose computer.

Another advantage that results from using a virtual machine is the case with which the computer can be ported on other machines. **The best example is with the Java language**, although there have been others, such as the original Pascal compiler, which generated code for a virtual machines called P-code. Usually, porting the compiler to another machine requires that the entire code generation process be rewritten. In the case of virtual machine, porting the compiler to another machine is a matter of resolving compatibility issues with the new compiler for the host architecture, that is the compiler that compiles the language being ported. Once a compiler has been ported to another architecture, it can be left largely the same as it was on the previous architecture, and it will still generate code that executes properly on VM. Where the real work lies in porting a VM language to another computer is in porting the VM, itself. However, virtual machines themselves are usually quite simple in nature. If that were not the case, they would be very slow.

Tips – The price that is paid these advantages spoken of so far is the lack of speed at run time; A virtual machine, sometimes called an interpreter, is always slower than a real machine.

**1.11.2 Pros and Cons of Both the Architecture**

|  |  |  |
| --- | --- | --- |
|  | Hardware CPU | Virtual Machine |
| Pros:  Cons: | Very, very fast by comparison  Generalized architecture and versatile  Low-level instruction set for doing low-level tasks  Complex and tedious to work with  Non-portable. Each CPU is unique  Not all CPU’s have the same features  Lower level  Complex instruction formatting  Requires some knowledge of the operating systems as well | Somewhat simpler  Portable, all VM features are standardized  Able to have features that are expensive in hardware  Higher-level instruction set  Very, very slow by comparison |

**1.12 LANGUAGE IMPLEMENTATION AND COMPILER DESIGN**

After knowing the introduction of computer language and compiler, one can conclude that implementation of particular language plays an important role in the designing of compiler (Translators) for that particular language. Let us discuss some common issues of computer languages implementation which always makes an impact in compiler designing.

**1.12.1 Data Types**

We all know the computer program produce results by manipulating data. An important factor in determining the case with which they can perform this task is how well the data types match the real-world problem space. It is, therefore, crucial that a language support an appropriate variety of data types and structures.

Data items can be viewed as follows:

a. Data items can be variable or constants or anonymous.

b. Variable or constants can be global or local.

c. Data items are “introduced” using a declaration statement.

d. Data items are having an “initial” value associated with them through a process known as initialization.

e. The value associated with a variable can be changed using an assignment operation.

Finally data types can be defined as follows:

a. The nature of the internal representation of the data item.

b. The interpretation placed on the internal representation.

c. As a result of (2) the allowed set of values for the item.

d. The operation that can be performed on it.

The process of associating a type with a data item is referred to as a data declaration.

**1.12.1.1 Categorization of Types**

We can divide the data types in four basic categories:

a. **Predefined and programmer-defined types** – Pre-defined types are types immediately available to the user. Programmer-defined types are types derived by the programmer.

b. **Scalar and Compound Types**. Scalar types define data items that can be expressed as single values. Compound types define data items that comprise several individual values.

c. **Discrete and Non-discrete Types**. Discrete types are data types for which each value (except its min. and max.) has a predecessor and a successor value. The opposite are non-discrete types.

d. **Basic and Higher-level Types**. Basic types are the standard predefined scalar types available for immediate use in any imperative programming language. Higher level types are made up of such basic types and existing higher level types.

Let us see predefined data types of Ada, C and Pascal:

Predefined Data Types

Ada C Pascal

Character Character char char

Integer Integer int integer

Real numbers Float float real

Logical type Boolean - Boolean

Character string String - text

Void - void -

Fig 1.16

**1.12.1.2 Attributes of Data Items**

A data item has a number of attributes:

a. **Address** – The Address (reference) of a data item is the physical location in memory (computer store) of that data item. The amount of memory required by a particular data item is dictated by its **type** that is integer or character.

For example in C:

int ab;

mean ab is identifier and memory required for it will be 2 byte

b. **Value** – The binary number stored at an address associated with a data item is its **value**. The required interpretation is dictated by the type of the integer.

Consider:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |

This might be interpreted as:

i. The Decimal integer `90’

ii. The Hexa decimal integer `5A’

iii. The Octal integer `132’

iv. The ASCII character (Capital) `Z’

c. **Operations** – The type of a data item also dictates the operations that can be performed on it. For example, numeric data types can have the standard arithmetical operations performed on them while string data types cannot.

d. **Names** – To do any thing useful with a data item, all the above must be tied together by giving data item an identifying name (also referred to as a label or identifier).

i. Names are usually made up of alphanumeric characters (and should be descriptive).

ii. While space is usually not allowed (except Algal 60).

iii. Some languages (Ada and C) allow underscores.

iv. Some languages restrict the number of characters.

v. Some allow any length but treat the first N characters as significant.

vi. Case may be significant in Algal-60, C and Modula 2, in Pascal and Ada it does not.

vii. Cannot use key/reserved words as names.

viii. Naming conventions.

**1.12.1.3 Data Declarations**

A data declaration introduces a data item with all its attributes. This is typically achieved by associating the item (identified by a name) with a data type.

Declarative examples

in Ada : Number : integer ;

Pascal : number : integer ;

and C: int number ;

In some imperative languages (notably Pascal) declarations must be presented in a certain order that is constants before variables. In declaration initialization is then the process of assigning an “initial” value to a data item upon declaration (not all imperative language support this, C and Ada do, Pascal does not).

Initialization examples in C:

int number = 2 ;

and in Ada:

Number : integer : = 2 ;

**1.12.1.4 Assignment**

Assignment is the process of associating a value with a data item. Usually achieved using an assignment operator.

Number : = 2 ; (Assignment in Ada)

number : = 2 ; (Assignment in Pascal)

number = 2 ; (Assignment in C)

Many imperative languages (Ada and Pascal, but not C) support the **concept of multiple assignment** where a number of variables can be assigned a value (or values) using a single assignment statement.

In Ada:

Number 1, Number 2 : = 2 ;

In case of Pascal, we do not have to assign the same value to each variable when using a multiple assignment statement:

number 1, number 2 : = 5, 4 ;

In Pascal, we can also write :

number 1, number 2 : = number 2, number 1 ;

which has the effect of “doing a swap”.

**1.12.2 Data Structures in High Level Programming Languages**

HLLs permit the programmer to declare and use data structures. In HLLs, the available basic types can be extended by adding higher level types. Higher level types are (usually user-defined) data types made up of basic types and other user-defined higher level types. There are some standard higher level types available in most imperative languages. These include:

a. Arrays

b. Strings

c. Enumerated types

d. Records/Structures

e. Unions

The programs can access parts of the data structures in terms of units meaningful to him, that is the 7th element of a single dimensioned array Y, etc. The compiler now has to develop and use a storage mapping to access the actual storage cells allocated to various elements of the data structures.

Let us see an example of computer language C:

Struct emp {

char name [10] ;

char sex ;

int id ;

} info [500] ;

enum day {Monday, Tuesday, Wednesday, Thursday, Friday}

int I, j ;

day today

info [i] . id = j ;

if (today = Monday)

{

=========

Statements ;

=========

}

Here, info is an array of structures. The reference info [i] id involves two different kinds of mappings. The first one is the homogenous mapping of an array reference. In the second mapping, the compiler has to provide for access to the field id from a collection of heterogenous field. Day is a programmer defined data type. A totally different mapping is involved here since the compiler must figure out how to represent the different values of the programmer defined type, and then develop an appropriate mapping.

**1.12.3 Lifetime of a Variable**

A variable’s lifetime is the period of time during which the variable exists during execution. Some variables exist during execution. Some variables exist briefly. Some are repeatedly created and destroyed. Others exist for the entire execution of a program. In this section, we will discuss two storage classes of variables: automatic variables and static variables.

An **automatic** variable’s memory location is created when the block in which it is declared is entered. An automatic variable exists while the block is active, and then it is destroyed when the block exited. Since a local variable is created when the block in which it is created is entered and is destroyed when the block is left, one can see that a local variable is an automatic variable.

A **static variable** is a variable that exists from the point at which the program begins execution and continues to exist during the duration of the program. Storage for a static variable is allocated and initialized once when the program begins execution. A global variable is similar to a static variable since a global variable exists during the duration of the program. Storage for the global variable is allocated and is initialized once when the declaration for the global variable is encountered during execution. Thus, both the global variable and the static variable have a history preserving feature that is they continue to exist and their contents are preserved throughout the life time of the program.

For most applications, the use of automatic variables works just fine. Sometimes however, we want a function to remember values between function calls. This is the purpose of a static variable. A local variable that is declared as static causes the program to keep the variable and its latest value even when the function that declared it is through executing. It is usually better to declare a local variable as static than to use a global variable. A static variable is similar to a global variable in that its memory remains for the lifetime of the entire program.

**However, a static variable is different from a global variable because a static variable’s scope is local to the function in which is it defined**. Thus, other functions in the program cannot modify a static variable’s value because only the function in which it is declared can access the variables.

In Fortran, variables are allocated statically. All variables are allocated storage before execution of the program begins. As consequence, when return to a procedure local variables still have last value at end of previous invocation.

In Pascal or C, when enter procedure any local variables are allocated and then deallocated when exit (called dynamic allocation of variables).

Block-structured languages (such as Pascal, C, Modula-2 etc) uses runtime stack to allocate space for local variables and parameters when enter a new unit (procedure, function etc) and that space is called an activation record. Note that a procedure may have several activation records on stack if called recursively. Even without recursion may have several distinct variables on stack with same name. Pascal and C uses three kinds of memory: Static (occupied by global variables), stack-based or automatic (occupied by parameters and local variables of procedures and functions), and heap-based or manually allocated. In ML, everything comes off of heap. But automatically allocated when needed and deallocated (by garbage collector) when no way of accessing it, Java is similar.

**1.12.4 Scope**

The scope of a program variable is the range of statements in which the variable is visible. A variable is visible in a statement if it can be referenced in that statement.

We can divide scope in two parts:

a. **Static Scope** – This kind of the scope is associated with the static text of the program. Static scope can be determined by looking at structure of program rather than execution. Languae like Pascal, Modula-2 and C use static scoping

An example of Pascal language is:

Procedure M ;

Var X : integer ;

Procedure subl;

begin [start of sub1]

------ x -----

end ; [end of sub1]

Procedure sub2 ;

Var X ; integer

begin [start of sub2]

======

end ; [end of sub2]

begin [start of M]

======

end ; [end of M]

Assume that a reference is made to a variable x in sub-program Sub1, but no declaration is found there. The search thus continuous in the static parent of sub1, M, where the declaration of x is found. Although C and C++ do not allow subprogram to be nested inside other sub-program definitions, they do have global variables. These variables are declared outside any sub-program definition. Local variables can hide these globals, as in Pascal.

b. **Dynamic Scope** – Dynamic scoping is based on the calling sequence of sub-programs, not on their spatial relationship to each other. This scope can be determined at run time only.

An example of Pascal is as follows:

Program ……………………….

Var A : integer ;

Procedure Y (B ; integer)

begin

----------- ;

B : = A + B ;

----------- ;

end ; [End of Y]

Procedure Z (-------------------) ;

Var A : integer ;

begin

------------- ;

Y (-------------) ;

end ; [end of Z]

begin [Start of main]

------------------

------------------

Z (-------------------) ;

end ; [End of main]

Now Big Question is that which variable with name A is used when Y is called from Z?

a. In case of static, clearly globally defined A.

b. In case of dynamic scoping, local A in Z, since declaration of A in Z is most recent.

**1.12.5 Binding**

A binding is an association, such as between an attribute and an entity or between an operation and a symbol. The time at which a binding takes place is called **binding time**.

Binding can be categorized into several types:

Some bindings are performed at run time are **called run time binding**, binding may be performed **at compile time** and it remains static throughout program execution. Sometime **programmers and translators also** can decide about the binding. Binding may be decided when **language design is implemented** and designer of the **language is defining the language**. Binding can also be done at **load time and link time**.

Let us consider the following C assignment statement:

int x ;

--------------

--------------

C = x + 10 ;

Some of the bindings and their binding times for the parts of this assignment statement are as follows:

a. The possible set of types for x and this bound is decided at language design time.

b. The type of x is bounded at compile tim.

c. Set of possible values of x, bonds at compile time.

d. Value of count, bounds at execution time with this statement.

e. Set of possible meaning for the operator symbol +, bound at language definition time.

f. Meaning of the operator symbol + in this statement, bonds at compile time.

g. Internal representation of the literal 10, bounds at compiler design time.

**1.12.6 Fundamentals of Sub-programs**

Sub-programs are the fundamental building blocks of programs and are, therefore, among the most important concepts in programming language design.

Let us see an example of C sub-program:

return type identifier (argument 1, argument 2, …..)

{

------------------

------------------

------------------

Statements

------------------

------------------

------------------

}

The first statement is called header of the C function and block is called body of function. **The parameters** in the sub-program header are called **formal parameters**. Sub-program call statements must include the name of the sub-program and a list of parameters to be bound to the formal parameters of the sub-program. These parameters are **called actual parameters**. The first actual parameter is bound to the first formal parameter and so forth, such parameters are called positional parameters.

**1.12.6.1 Parameter-Passing Methods**

Parameter-passing methods are the ways in which parameters are transmitted to and/or from called sub-programs. A variety of methods has been developed by language designers.

Let us see some of them:

a. **Parameter Passing-By-Value** – when a parameter is **passed by value**, the value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable in the sub-program. It can be understood by the following example:

Caller Call Callee

add (2,3) Actual transfer int add (int a, int b)

of 2 and 3 {

int c = a + b;

return c;

}

Fig. 1.17

Pass-by-value is normally implemented by actual data transfer. The main disadvantage of the pass-by-value method, if physical moves are done is that additional storage is required for the formal parameters, either in the called sub-program or in some area outside both the caller and the called sub-program. The storage and the move operations can be costly if the parameter is large, such as long array.

**Pass-By-Reference** – In case of Pass-By-Reference method, we pass reference of actual parameters to the formal parameters. The actual parameters are shared with the called sub-program.

Tips – However, there are several disadvantages to the pass-by-reference method. First, access to the formal parameters will most likely be slower because one more level of indirect addressing is needed than when data values are transmitted. Another serious problem of pass-by-reference is that aliases can be created. This is expected because pass-by-reference makes access paths available to the called sub-programs.

Caller Callee

Passing of References void action (int \*x, int yy)

action (sa, sb) {

-----------

-----------

}

Fig. 1.18

**Exercise**

1. What are phases of a compiler? Explain the function of each phase in brief.

2. Symbol table is necessary for compiler construction, justify your statement with example.

3. What is a cross compiler? How is boot-strapping of a compiler done to a second machine?

4. What is a translator? Discuss the role of various phases of the compiler in the translation of source program to object code.

5. Suppose you have a working C compiler on machine A. Discuss the steps you would take to create a working compiler for another language C on machine B.

6. Discuss the action taken by every phase of the compiler on the following string:

A = B \* C + D/E

7. Write the difference between syntax and semantics of the languages, also give the example in C language.

8. Analyze the programs of any language on the basis of their different programming structure.

9. Discuss the advantages and disadvantages for single ad multipass compilers.

10. Discuss the role of machine architecture in compiler design, also give the pro and cons of virtual machine and real machine architecture.

**CHAPTER TWO**

**INTRODUCTION TO FINITE AUTOMATA AND REGULAR EXPRESSION**

**This Chapter considers the following:**

2.1 Introduction to Automata

2.2 Alphabets

2.3 Strings

2.4 Languages

2.5 Deterministic Finite Automata

2.6 Non-determinism

2.7 The Language of NFA

2.8 Transformation of NFA to DFA

2.9 NFA with e-transitions

2.10 Acceptance of A string by NFA with e-transitions

2.11 Conversion of NFA with Ԑ-transitions to NFA without Ԑ-transitions

2.12 Minimization of DFA’s

2.13 Regular Expression

2.14 Comparative Study of Regular Expression, Regular Sets and Finite Automata

2.15 Construction of FA for Regular Expression

**2.1 INTRODUCTION TO AUTOMATA**

These are machines (automata) that mimic a computer by providing a medium, which executes a finite number of instructions sequentially according to an algorithm, accepting valid input and producing an output if the input is accepted. All these machines have an input tape, a tape reading head, a register recording the state of the machine which is one of the finite states in which it can exist, and a set of instructions that govern the transition from one state to another.

P0, P1, P2, P3, P4, P5 are states in finite control system, x and y are input symbols. At regular interval, the automation reads one symbol from the input tape and then enters in a new state that depends only on the current state and the symbol just read. After reading an input symbol, reading head moves one square to the right on the input tape, so that on the next move, it will read the symbol in next tape square. This process is repeated again and again. The automation then indicates its an approval or disapproval of what it has read by the state it is at the end.

Tips – If it winds up in one of a set of final states the input strings is considered to be accepted. The language accepted by the machine is the set of strings, it accepts.

x y x y y x y ………

INPUT TAPE

READING HEAD

P0

FINITE P5 P1

CONTROL

SYSTEM P4 P2

P3

Fig. 2.1

**2.2 ALPHABETS**

Everything in mathematics is based on symbols. This is also true for automata theory of computation. The symbols are generally letters and digits. “Alphabets are defined as a finite set of symbols”. An example of an alphabet is a set of decimal numbers: ∑ = {0,1,2,3,4,5,6,7,8,9}

Here ∑ is representation for the alphabet.

Another example is a binary alphabet that is:

∑ = {0, 1}

Similarly ∑ = {A, B, … Z} is a alphabet

∑ = {a, b, … z} is also a alphabet}

**2.3 STRINGS**

“A string (or sometimes word) is a finite sequence of symbols selected from some alphabet”. For example, if ∑{a, b} is an alphabet then *abab* is string over alphabet ∑. A string is generally denoted by *w*.

The “length” of the string is denoted by |*w*| and it is, the number of positions for the symbol in the string. For instance string: *w* = 01101 has length

|*w*| = 5

Tips – It is common to say that the length of the string is “the number of symbols” in the string; this statement is generally accepted but not strictly correct. Thus, there are only two symbols, 0 and 1, in the string 01101, but there are five positions for symbols, and its length is 5.

The “empty” string is the string with **zero occurrence of symbols**. This string is represented by ϵ, or e or λ, or γ  **but we will follow** ϵ i**n** t**his particular book**.

The set of strings, including the empty, over an alphabet ∑ is denoted by ∑\*, that is suppose ∑ = {0, 1} then ∑\* is a set containing empty and all combinations of 0’s and 1’s. Thus:

∑\* = {ϵ, 0, 01, 10, 000, 010, 0000, 1111000 …}

Similarly {0}\* = {ϵ, 0, 00, 000 …}

{1}\* = {ϵ, 1, 11, 111 …}

Note that ϵ is in ∑\*, regardless of what alphabet ∑ is

**That is Ԑis only string whose length is 0**.

If ∑ = {0, 1}, then ∑1 = {0, 1}, ∑2 = {00, 01, 10, 11},

∑3 = {000, 001, 010, 011, 100, 101, 110, 111}

And so on. Note that there is slight confusion between ∑ and ∑1. The formal is an alphabet, its members 0 and 1 are symbols. The latter is a set of strings; its members are the strings 0 and 1, each of which is of length 1. We shall not try to use separate notations for the two sets, relying on context to make it clear whether {0, 1} or similar sets are alphabets or set of strings.

Sometimes, we wish to exclude the empty string from the set of strings. The set of non-empty strings from alphabet ∑ is denoted by ∑+. Thus, two appropriate equivalence are:

∑+ = ∑1 ᴗ ∑2ᴗ ∑3ᴗ …

∑\* = ∑+ ᴗ {ϵ}

**Concatenation of strings** – Let w1 and w2 be two strings. Then w1 w2 denotes the concatenation of w1 and w2, that is, the string formed by making a copy of w1 and followed it by a copy of w2.

if *w1 = abc*

*w2 = xyz*

*t*hen *w1w2 = abc xyz*

In the set form string w2 is defined as:

*w2 =* ϵ *for i = 0*

*wi-1 w for i ≥ 1*

that is if w = aba

then *w3 = (aba)3*

*w3 = aba aba aba*

*= (aba)2 aba*

**Reverse of the String** – Reverse of the string can be achieved by flipping over last symbols:

if w = abc

then wR = cba, where wR is reverse of w, |w| = |wR|

**2.4 LANGUAGES**

“A set of strings all of which are chosen from some ∑ \*, where ∑ is a particular alphabet, is called a language”. If ∑ is an alphabet, and L ∑ \*, then L is said to be language over alphabet ∑.

Notice that a language over ∑ need not include strings with all the symbols of ∑, so once we have established that L is a language over ∑, we also know it is a language over any alphabet which is a superset of ∑.

An example is English, where the collection of legal English words is a set of strings over the alphabet that consists of all the letters. Another example is C or any other programming language, where the legal programs are a subset of the possible strings that can be formed from the alphabet of the language. This alphabet is a subset of the ASCII characters. The exact alphabet may differ slightly among different programming languages, but generally includes the upper-and lower-case letters; the digits, punctuation and mathematical symbols.

However, there may be some other languages that appear when we study automata. Some examples are as follows:

a. The language of all strings consisting of n 0’s followed by n 1’s, for some n ≥ 0: {ϵ, 01, 0011, 000111, …}.

b. The set of strings over 0’s and 1’s with equal number of each: {ϵ, 01, 10, 0011, 0101, 1001 …}.

c. The set of binary numbers whose value is a prime: {10, 11, 101, 111, 1011, …}.

d. ∑\* is a language for any alphabet ∑.

e. ɸ,the empty language, is a language over any alphabet.

f. {ϵ}, the language consisting of only the empty string, is also a language over any alphabet. Notice that ɸ ≠ {ϵ}; the former has no strings and latter has one string.

The only important constraint on what can be a language is that all alphabet are finite. Thus, languages, although they can have an infinite number of strings, are restricted to consist of strings drawn from one fixed, finite alphabet.

**2.4.1 Concatenation of Languages**

Concatenation of languages can be defined as:

If L1 and L2 are two languages, their concatenation

*L = L1, L2 where L = {w|w = xy, x ϵ L1, y ϵ L2}*

That is, the string in L1 L2 are formed by choosing a string in L1 and followed by a string in L2, in all possible combinations.

**2.4.2 Languages in Set Forms**

It is very common to describe a language using a “set-form”:

**{w/some logical view about w}**

This expression is read “the set of words *w* such that (whatever is said about *w* to the right of the vertical bar)”. For example:

a. {w/w consists an equal number of 0’s and 1’s}

b. L = {w/w is a binary integer that is prime}

It is common to replace w by some expression with parameters and describe the strings in the language by stating conditions on the parameters. Here are some examples: {anbn/n ≥ 1}. Read “the set of *a* to the *n*, *b* to the *n* such that n is greater than or equal to 1”, this language consists of the strings {*ab, aabb, aaabbb, …*}. Notice that as with alphabet, we can raise a single symbol to a power n in order to represent n copies of that symbol.

**2.5 DETERMINISTIC FINITE AUTOMATA**

We know that computer is deterministic, by which we mean that, on reading one particular input instruction, the machine converts itself from the state it was into some particular other state, where the resultant state is completely determined by the prior state and the input instruction. Some sequence of instruction may lead to success and some may not. Success is entirely.

**Finite automata is called “finite” because number of possible states and number of letter in the alphabet are both finite, and “automation” because the change of the state is totally governed by the input. It is deterministic, since, what is next is automatic not will-full, just as the motion of the hands of the clock is automatic, while the motion of hands of a human is presumably the result of desire and thought.**

When we analyse above discussion mathematically the following formal version is obtained.

A deterministic finite automata is a quintuple

M = (*Q, ∑, δ,* *q*0, *F*)

Where

Q : is a non-empty finite set of states present in the finite control (q0, q1, q2, …)

∑ : is a non-empty finite set of input symbols which can be passed to the finite state machine (*a, b, c, d, e, …*)

q0 : is a starting state, one of the state in Q

*F* : is a non-empty set of final states or accepting states, set of final states belongs to *Q*

δ : is a function called transition function that takes two arguments ---- a state and a input symbol, it returns a single state

Let `q’ is the state and `a’ be input symbol passed to the transition function as:

δ(q, a) = q’

q’ is the output of the function i.e., a single state only, q’ may be q.

**Some people prefer to call this just a finite acceptor because its sole job is to accept certain input strings and rejects others. It does not do anything like printing machine. Some time, it is also referred as language recognizer because FA merely recognizes whether the input string is in the language as we can also recognize that somebody is speaking *Russian or English* without necessarily understanding what it means.**

**2.5.1 Processing of Strings by DFA**

Suppose a1, a2, a3, …, an is a sequence of input symbols. We start out with deterministic finite automata having q0, q1, q2, q3, … qn states where q0 is the initial state and qn is the final state and transition function and processed as

δ(q0, a1) = q1

δ(q1, a2) = q2

δ(q2, a3) = q3

⁞ ⁞

⁞ ⁞

δ(q0, a1) = q1

Input a1, a2, …, an is said to be “**accepted**” since qn is the number of the final states, and if not then it is “**rejected**”. By adopting this approach we can define the language of a particular DFA, and can also check that particular string is accepted or rejected by given FA.

Tips – “So a string is said to be accepted by the given FA only when it is processed by transition function d in such a manner that it ends at the final state”.

**2.5.2 Simpler Notations for DFA’s**

There are two preferred notations for describing automata:

a. Transition Diagram

b. Transition Table

Transition Diagram Notations – A transition diagram for DFA, M = (*Q, ∑, δ, q0, F*) is a graph defined as follows:

i. For each state in Q there is a node represented by the circle.

ii. For each state q in Q and each input symbol a in ∑, let δ(q, a) = P. Then transition diagram has an arc from node q to node P, labeled a. If there are several input symbols that cause transition from q to P, then the transition diagram can have one arc, labeled by the list of these symbols as:

q a P or q a,b P or q a P

b

Fig. 2.2

iii. If any state q in Q is the starting state then it represented by the circle with arrow as:

q

Fig. 2.3

iv. Nodes corresponding to accepting states (those are in F) are marked by a double circle. States not in *F* have a single circle.

Fig. 2.4

**Transition Tables** – A transition table is a Conventional, tabular representation of a function like δ that takes two arguments and returns a state. The rows of the table correspond to the states, and the columns correspond to the input. The entry for one row corresponding to state q and the column corresponding to input a is the state δ(q, a). For example:

δ/∑ a b

q0 q1 q2

q1 q2 q0

\*q2 q2 q2

Fig. 2.5

q0 is the starting state and q2 is the final state,

Q = {q0, q1, q2}

F = {q2}

δ(q0, a) = q1, δ(q0,b) = q2

δ(q1, a) = q2, δ(q1,b) = q0

δ(q2, a) = q2, δ(q2,b) = q2

Let us draw the transition diagram for the transition table shown in Fig. 2.6

b

q0 a q1

b a

q2

a,b

Fig. 2.6

So clearly transition diagram can be find from transition table and vice versa is also true.

**2.5.3 The Language of a DFA**

It have been explained informally that DFA defines a language: The set of all strings that result in a sequence of state transition from start state to an accepting state. Now we can define the language of a DFA M = (Q, ∑, δ, q0, F). This language is denoted by L (M), and defined by

*L(M) = {w|δ (q0, w) is in F}*

\*δ (extended transition function) will be explained afterwards.

That is the language of M is the set of strings (w) that take the start q0 to one of the accepting states.

*Tips – If L is L(M) for some deterministic finite automata, then we say L is a regular language.*

Now for practical understanding of FA, let us discuss some examples:

**Example 2.1 – Design a FA that accepts set of strings such that every string ends in 00, over alphabet**

{0, 1} i.e., ∑ = {0, 1}

Solution – Let M = (Q, ∑, δ, q0, F) be the DFA)

q0 : initial state

∑ : {0, 1} is given

So according to the problem, we have to design a FA which accepts strings w = w’00

Where w’ can be any combination of 0’s and 1’s for example 011100, 111100, 00, 100, 01010100 etc.

Now first we will decide the approach to design FA. It is not an easy task to think FA as a whole. So first we have to fulfill minimum condition i.e. every string end in 00.

0

Start q0 0 q1 0 q2

Fig. 2.7