BACHELOR OF COMPUTER APPLICATIONS

SEMESTER 4

DCA2203

SYSTEM SOFTWARE

Unit 7

Interpreter

Table of Contents

1. INTRODUCTION

A program that executes instructions written in a high-level or assembly language by translating that source code to object code is called Language Translator. There are two ways to run programs written in a high- level language. The most common is to compile the program; the other method is to pass the program through an interpreter.

In this unit, we are discussing Interpreter and compiler and their differences. Also, we will discuss scanning, symbol table, parsing expressions and assignments, and control statements. In this unit's last section, we discuss a simple Interpreter Design.

1.1 Learning Objectives:

After Studying this unit, Learners are able to

- *Define an Interpreter and a Compiler.*
- *Differentiate between compiler and Interpreter.*
- *Describe about Scanning and Parsing expressions.*
- *Explain about a symbol table*
- *Explain about control statements.*
- *Describe a simple Interpreter design.*

2. INTRODUCTION TO INTERPRETER

An Interpreter is a language translator who converts a high-level language to a low-level language (machine language). It is similar to a compiler, but during compilation, the whole source program (a program written in high-level language) is converted into the object program (a program written in low-level language), but an interpreter is a line-by-line translation.

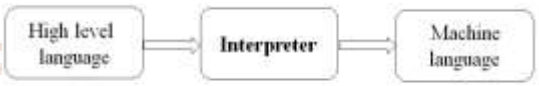


Figure 7.1: Interpreter

There are lots of definitions available for the interpreter. These definitions will give an idea about interpreters and their functions. Interpreters take the HLL (High-Level Language) program as input, examine each instruction, and execute the equivalence machine language program. This process is called interpretation high-level as shown in figure 7.1. An interpreter occupies *less memory space; it is cheaper and suitable for smaller systems. It is slower than the compiler.* Interpreters have a quick turn-around time, which is useful for interactive systems. There is no edit, compile, or test cycle. They are portable.

Strategies of an Interpreter

It can work in three ways:

- Directly run the source code to generate the result.
- Execute the intermediate code created by converting the source code.
- Generating precompiled code using an internal compiler. After that, run this precompiled code.

Advantages

- Carries out line by line. Debugging is, therefore simple.
- Because there is no intermediate code, memory is used effectively.

Disadvantages:

- More time is spent on execution.

3. OVERVIEW OF COMPILER

The compiler is a translator. It translates the program written in a high-level language into an object language, a low-level language (machine language). Executing a program written in a high-level programming language is a two-step process, as illustrated in Figure 7.2. The source program must first be compiled and translated into the object program. Then the resulting object program is loaded into memory and executed.

Figure 7.2: Compilation and Execution

With machine language the communication is direct with a computer in terms of bits, registers, and very primitive machine operations. Since a machine language program is nothing more than a sequence of 0's and 1's, programming a complex algorithm in such a language is tedious, and there is a chance of making mistakes. Perhaps the most serious disadvantage of machine language coding is that all operations and operands must be specified in numeric code. Not only is a machine language program cryptic, but it also may be impossible to modify conveniently.

So, to overcome these problems, there has been an evolution of new programming languages. A high-level programming language allows a programmer to express algorithms in a more natural notation and avoids many of the details of how a specific computer functions.

A high-level programming language simplifies the programming task but also introduces some problems. The most obvious is that we need a program to translate the high-level language into a language the machine can understand.

4. PHASES OF COMPILER

A compiler takes a source program as input and produces an equivalent sequence of machine instructions as output . This process is so complex that it is not reasonable to consider the compilation process as occurring in one step, either from a logical point of view or from an implementation point of view. For this reason, it is customary to partition the compilation process into a series of sub-processes called phases, as shown in Figure 7.3.

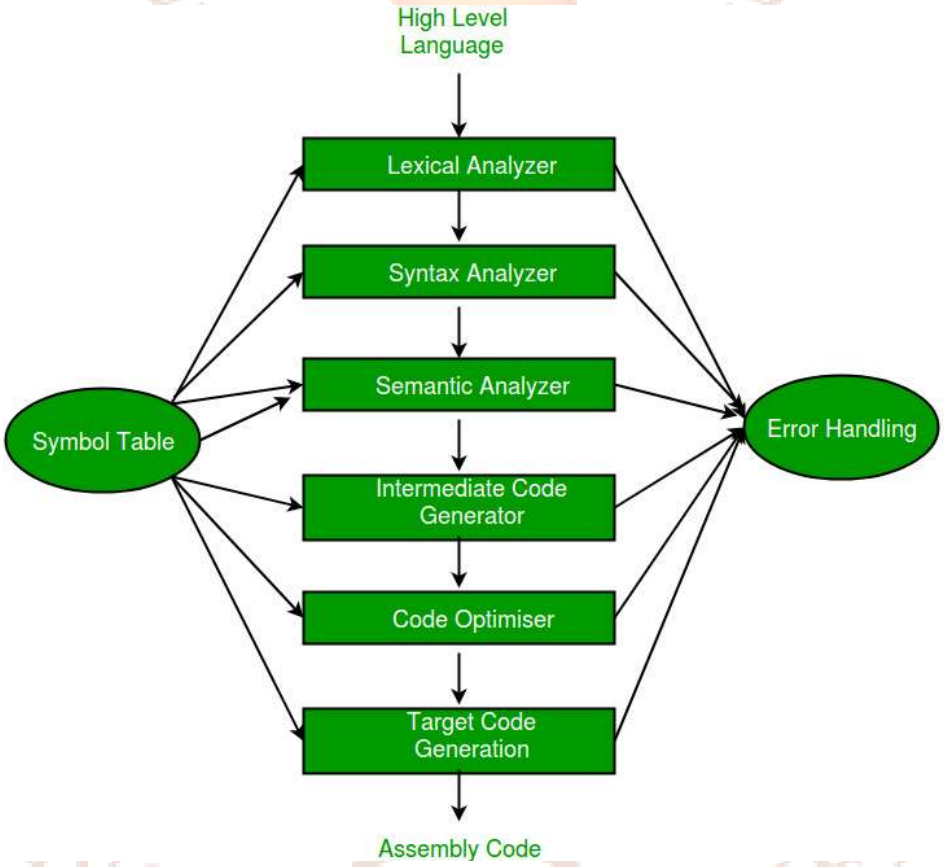


Figure 7.3: Phases of a compiler

A phase is a logically cohesive operation that takes as input one representation of the source program and produces as output another representation.

The first phase, called the *lexical analyzer or scanner*, separates characters of the source language into groups that logically belong together; these groups are called tokens. The usual tokens are keywords, such as DO or IF identifiers, such as X or NUM; operator symbols, such as < = or +, and punctuation symbols, such as parentheses or commas. The output of the

lexical analyzer is a stream of tokens, which passes to the next phase, the syntax analyzer, or parser. The tokens in this stream can be represented by codes that may regard as integers. Thus, DO might be represented by 1, + by 2, and "identifier" by 3. In the case of a token like 'identifier", a second quantity, telling which of those identifiers used by the program is represented by this instance of token "identifier," is passed along with the integer code for "identifier."

The *syntax analyzer* groups tokens together into syntactic structures. For example, the three tokens representing A + B might be grouped into a syntactic structure called an expression. Expressions might further be combined to form statements. Often the syntactic structure can be regarded as a tree whose leaves are the tokens. The interior nodes of the tree represent strings of tokens that logically belong together.

The *intermediate code generator* uses the structure produced by the syntax analyzer to create a stream of simple instructions. Many styles of intermediate code are possible. One common style uses instructions with one operator and several operands. These instructions can be viewed as simple macros like the macro ADD2. The primary difference between intermediate and assembly codes is that the intermediate code need not specify the registers to be used for each operation.

Code Optimization is an optional phase designed to improve the intermediate code so that the ultimate object program runs faster and takes less space. Its output is another intermediate code program that does the same job as the original but perhaps in a way that saves time and space.

The final phase, *code generation*, produces the object code by deciding on the memory locations for data, selecting the code to access each datum, and selecting the registers in which each computation is to be done. Designing a code generator that produces truly efficient object programs is one of the most difficult parts of compiler design, practically and theoretically.

Table-Management, or bookkeeping, a portion of the compiler keeps track of the names used by the program and records essential information about each, such as its type (integer, real, etc.). The data structure used to record this information is called a Symbol table.

The Error Handler is invoked when a flaw in the source program is detected. It must warn the programmer by issuing a diagnostic and adjusting the information being passed from phase to phase so that each phase can proceed. Compilation should be completed on flawed programs, at least through the syntax-analysis phase, so that as many errors as possible can be detected in one compilation. The table management and error-handling routines interact with all compiler phases. .

5. LEXICAL ANALYSIS

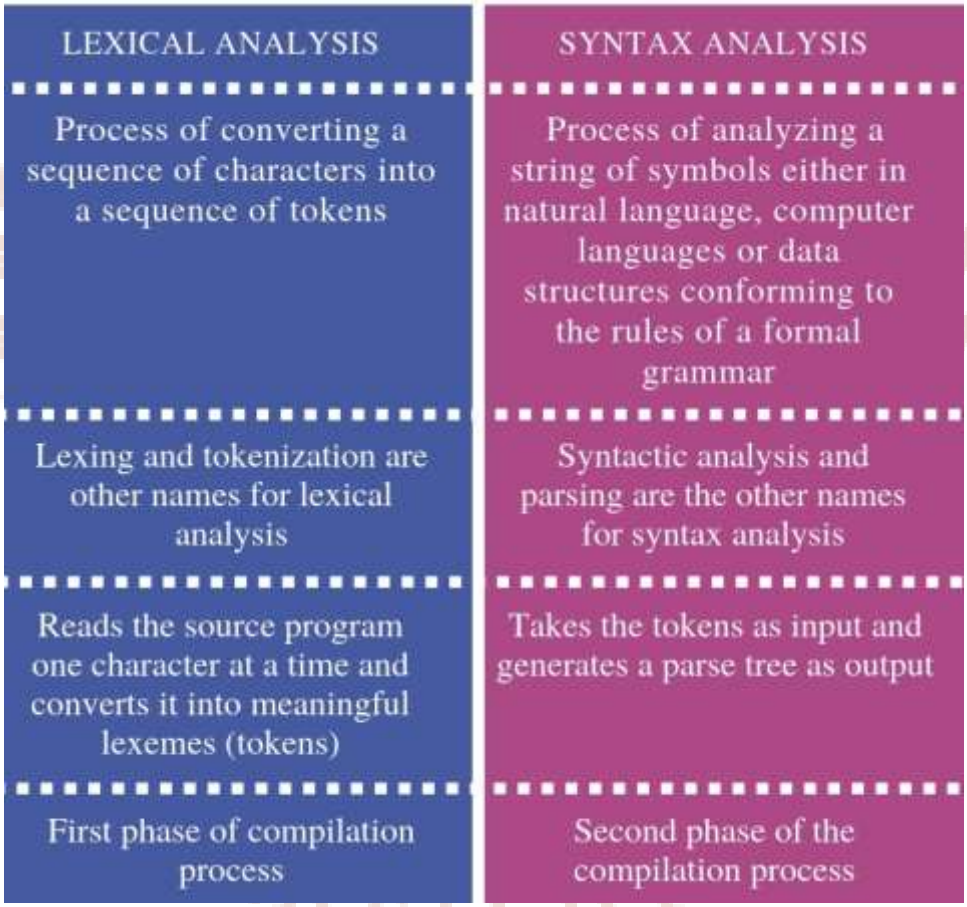
The lexical analyzer is the interface between the source program and the compiler. The lexical analyzer reads the source program one character at a time, carving the source program into a sequence of atomic units called tokens. Each token represents a sequence of characters that can be treated as a single logical entity. Identifiers, keywords, constants, operators, and punctuation symbols such as commas and parentheses are typical tokens. There are two kinds of tokens: specific strings, such as a semicolon and classes of strings, such as identifiers, constants, or labels.

The lexical analyzer and the syntax analyzer are often grouped into the same pass. In that pass, the lexical analyzer operates either under the control of the parser or a co-routine with the parser. The parser asks the lexical analyzer returns to the parser a code for the token that it found. If the token is an identifier or another token with a value, the value is also passed to the parser. The usual method of providing this information is for the lexical analyzer to call a bookkeeping routine which installs the actual value in the symbol table if it is not already there. The lexical analyzer then passes the two components of the token to the parser. The first is a code for the token type (identifier), and the second is the value, a pointer to the place in the symbol table reserved for the specific value found.

6. SYNTAX ANALYSIS

The parser (syntax analyzer) has two functions. It checks that the tokens appearing in its input, which is the output of the lexical analyzer, occur in patterns permitted by the specification for the source language. It also imposes on the tokens a tree-like structure that is used by the subsequent phases of the compiler called a parse tree.

The second aspect of syntax analysis is to make explicit the hierarchical structure of the incoming token stream by identifying which parts of the token stream should be grouped.



7. 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 Three-Address Code source program.

Three-Address Code

One popular intermediate language type the "three-address code". A typical three-address code statement is

A: = B op C

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

8. CODE OPTIMIZATION

Object programs that are frequently executed should be fast and small.

The output of the intermediate code generator is subjected to changes in a step of some compilers in an effort to create an intermediate-language version of the source program, from which a faster or smaller object-language program can finally be generated. This stage is frequently referred to as the optimization phase.A good optimizing compiler can improve the target program by perhaps a factor of two in overall speed compared to a compiler that generates code carefully but without using specialized techniques, generally referred to as code optimization. There are two types of optimizations used:

- Local Optimization
- Loop Optimization

9. 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 the 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. Therefore, a good code generator would attempt to utilize these registers as efficiently as possible. This aspect of code generation, called register allocation, is particularly difficult to do optimally.

1. Cross compilers

   They create platform-specific executable machine code for a different platform than the one the compiler operates on.

2. Bootstrap Compilers

   These compilers must be compiled since they are written in a programming language.

3. Source to source/transcompiler

   These compilers convert the source code of one programming language. into the source code of another programming language.

4. Decompile

   In essence, it isn't a compiler. It just functions as the compiler in reverse. The machine code is changed into high-level language.

Applications and Uses of Compilers

- Makes the code more platform independent.
- Removes any syntactic and semantic errors from the code.
- Create code files that can be executed.
- Converts the code between different languages.

Self-Assessment Questions - 1

1. Program which executes instructions written in a high-level language by translating that source code to object code is called _____ .
2. Compiler is a Language Translator. (True/False).
3. The intermediate code generation phase transforms parse tree into an intermediate language representation of the source program called " _____ ".
4. Lexical analyzer converts the source program into a sequence of atomic units called "_____ ".
5. The syntax analyzer groups tokens together into syntactic structures called "_____" .

10. DIFFERENCE BETWEEN COMPILER AND INTERPRETER

A Compiler and Interpreter both carry out the same purpose – convert a high-level language (like C, Java) instructions into binary form, which is understandable by computer hardware. They are the software used to execute the high-level programs and codes to perform various tasks. Specific compilers/interpreters are designed for different high-level languages. However, both compiler and interpreter have the same objective, but they differ in the way they accomplish their task i.e., convert high-level language into machine language. In this section, we will talk about the basic working of both and distinguish the basic difference between a compiler and an interpreter.

Compiler

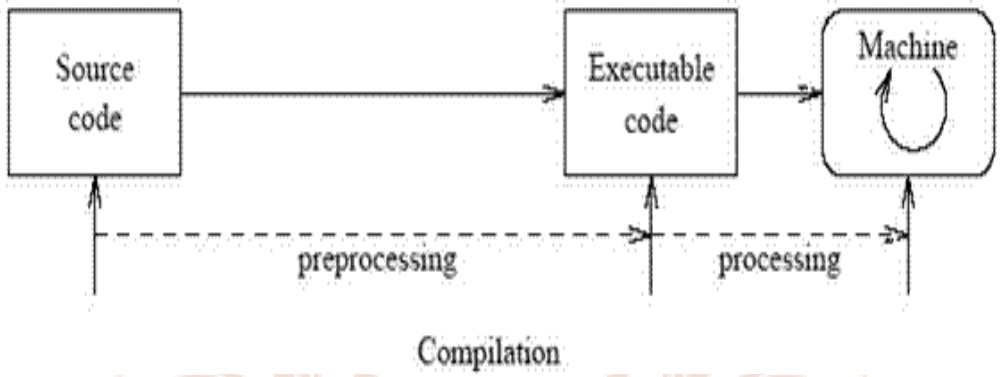


Figure 7.4: Process of compilation

Process of a Compilation

A compiler is a piece of code that translates the high-level language into machine language. When a user writes code in a high-level language such as Java and wants it to execute, a specific compiler that is designed for Java is used before it will be executed. The compiler scans the entire program first and then translates it into machine code which will be executed by the computer processor, and the corresponding tasks will be performed.

Shown in figure 7.4 is a basic outline of the compilation process; here program written in higher-level language is known as the source program, and the converted one is called an object program.

Interpreter

Interpreters are similar to from compilers. They also convert the high-level language into machine-readable binary equivalents. Each time an interpreter gets a high-level language code to be executed, it converts the code into an intermediate code before converting it into the machine code. Each part of the code is interpreted and then is executed separately in a sequence, and an error is found in a part of the code. It will stop the interpretation of the code without translating the next set of codes.

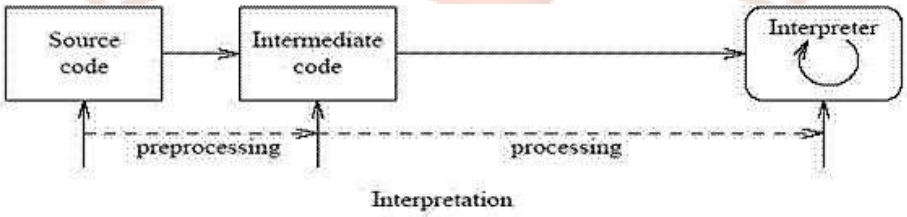


Figure 7.5: Process of Interpretation

Outlining the basic working of the interpreter above figure 7.5 represents that first, a source code is converted to an intermediate form, and then executed by the interpreter.

The main differences between compiler and interpreter are listed below:

- The interpreter takes one statement then, translates it and executes it, and then takes another statement. While the compiler translates the entire program in one go and then executes it.
- The Compiler generates the error report after the translation of the entire page, while an interpreter will stop the translation after it gets the first error.
- The Compiler takes a larger amount of time in analyzing and processing the high-level language code comparatively, interpreters take lesser time in the same process.
- Besides the processing and analyzing time, the overall execution time of a code is faster for the compiler relative to the interpreter

11. SCANNING

During scanning, the Interpreter scans the complete program and breaks the character stream into *tokens* (words). Identifiers, keywords, constants, operators, and punctuation symbols such as commas and parentheses are typical tokens. There are two kinds of tokens: specific strings, such as a semicolon and classes of strings, such as identifiers, constants, or labels. Figure 7.6 shows the structure of a typical interpreter.

If the token is an identifier or another token with a value, the value is also passed to the parser. The usual method of providing this information is for the lexical analyzer to call a bookkeeping routine which installs the actual value in the symbol table if it is not already there. The lexical analyzer then passes the two components of the token to the parser. The first is a code for the token type (identifier), and the second is the value, a pointer to the place in the symbol table reserved for the specific value found.

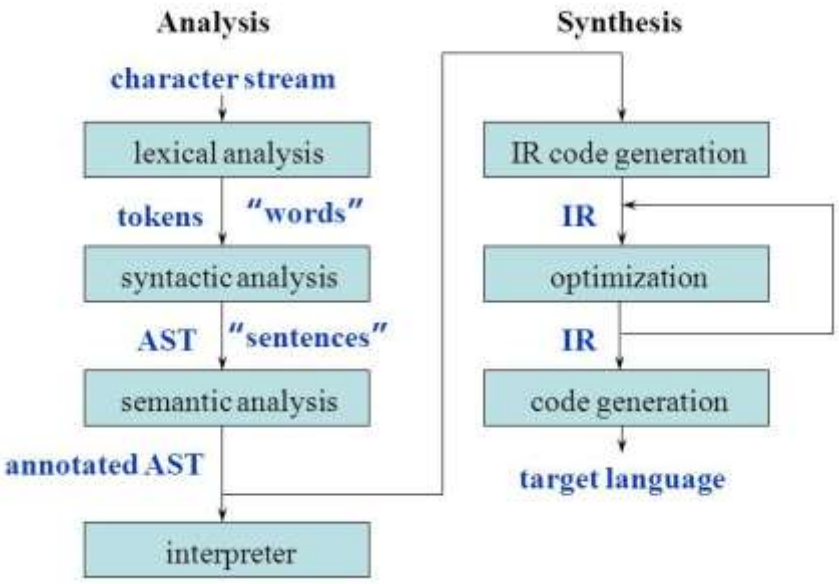


Figure 7.6: Structure of a typical Interpreter

Figure 7.7 shows the interaction between the scanner and the Parser.

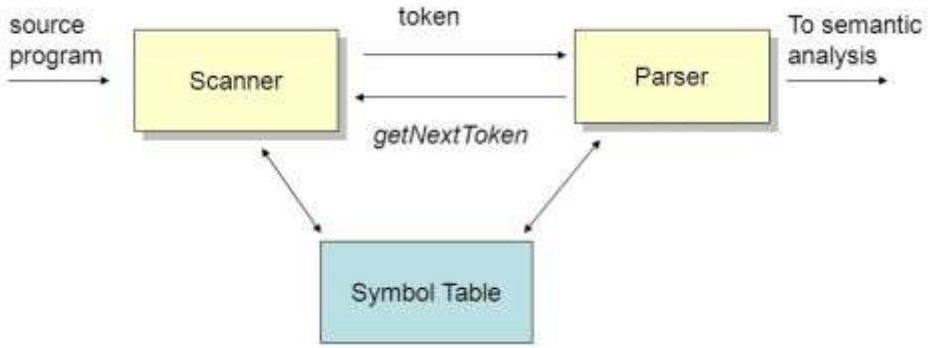


Figure 7.7: Interaction between scanning and Parsing

The token can be defined as a meaningful group of characters over the character set of the programming language, like identifiers, keywords, constants, and others.

Role of Scanner

The main task is to read the input character and procedure as an output sequence of tokens that the parser uses for syntax analysis upon receiving a"get next token" command from the parser; the lexical analysis reads the input characters until it can identify the next token. Its secondary tasks are

1. Stripping out from the source program comments and white space in the form of blank, tab, and new line characters.
2. Correlating error messages from the compiler with the source program.

Self-Assessment Questions - 2

6. A Runtime environment quickly compiles only the needed pieces of the code, which is called "_____".
7. Interpreter scans the complete program and breaks the character stream into tokens; this process is called "_____".
8. Input given to a parser is "_____".

12. SYMBOL TABLE

An essential function of an Interpreter is to record the identifier used in the source program and collect information about the various attributes of each identifier. A symbol table is a data structure containing a record for each identifier, with a field for the attribute. The data structure allows us to find the record for each identifier and to store or retrieve data from that's record quickly. When an identifier in the source program detected the lexical analyzers, the identifiers are entered in thesymbol table.

A Symbol table contains all the information that must be passed between different phases of a compiler/interpreter. A symbol (or token) has at least the following attributes:

- Symbol Name
- Symbol Type (int, real, char ...)
- Symbol Class (static, automatic, cons...)

Symbol tables are typically implemented using hashing schemes because good efficiency for the lookup is needed. The classification of symbol tables as:

- Simple
- Scoped

Simple symbol tables have:

1) only one scope and
2) only "global" variables.

The complication in simple tables involves languages that permit multiple scopes. C permits, at the simplest level, two scopes: global and local (it is also possible to have nested scopes in C)

Uses of Symbol Table

An object file will contain a symbol table of the identifiers that are externally visible. When linking different object files, a linker will use these symbol tables to resolve any unresolved references.

A symbol table may only exist during the translation process, or it may be embedded in the output of that process for later exploitation, for example, during an interactive debugging session or as a resource for formatting a diagnostic report during or after the execution of a program.

If the symbol table has been stripped or cleaned out before being converted into an executable, tools will find it harder to determine addresses or understand anything about the program.

At that time of accessing variables and allocating memory dynamically, a compiler should perform many works, and as such, the extended stack model requires the symbol table.

13. PARSING EXPRESSION AND ASSIGNMENT

A parser (also called syntax analyzer) is a program which receives valid tokens and checks them against the grammar and produces valid parse trees; otherwise generates syntactical errors. It can be said that the parser generates valid syntactical constructs as per the grammar of the source language from the valid tokens received from the lexical analyzer and generates syntax errors otherwise.

The parser has two functions. It checks that the tokens appearing in its input, which is the output of the lexical analyzer, occur in patterns permitted by the specification for the source language. It also imposes on the tokens a tree-like structure used by the subsequent phases of the compiler. The second aspect of syntax analysis is to make explicit the hierarchical structure of the incoming token stream by identifying which parts of the token stream should be grouped.

The syntax analyzer does syntax analysis. It groups the tokens of the source program into grammatical phrases that are used by the compiler to generate output. This is represented by the parse tree (hierarchical structure). The graphical representation for derivations that filters out the choice regarding the replacement order is called the parse tree.

The figure 7.8 shows the parse tree for the assignment statement

'Position: = Initial + rate*60'.

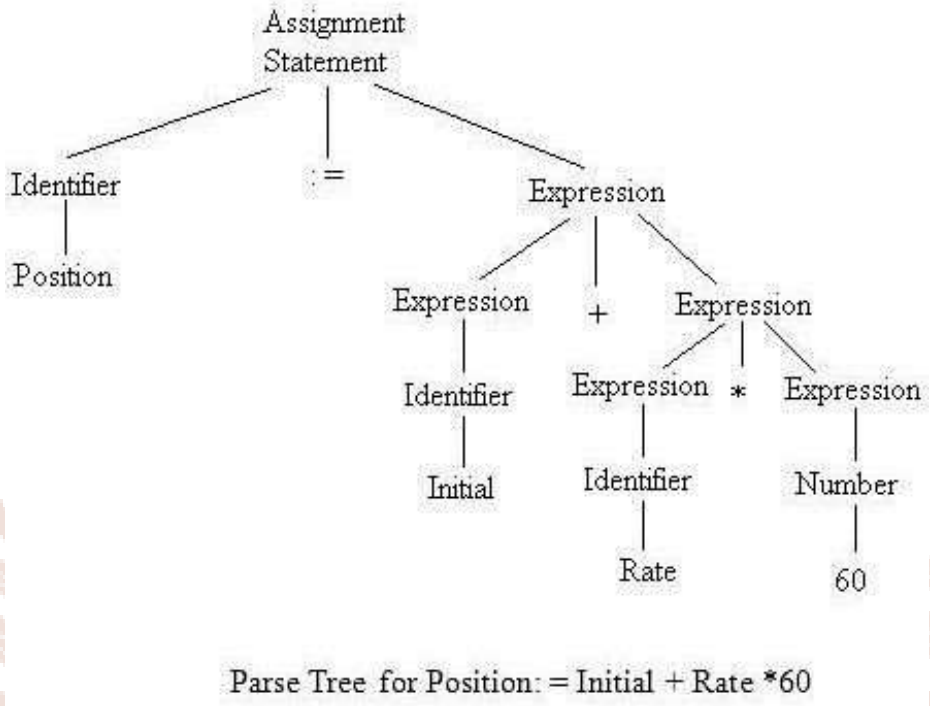


Figure 7.8: Parse Tree Representation

Parsers can be classified into two broad categories, namely,

1. Top-Down Parsers
2. Bottom-up Parsers.

Top-Down Parsers

These are the parsers that constructs the parse tree from the root to the leaves in preorder for the given input string. In simpler terms, the construction of the parse tree is done from top to bottom of the tree. Here the starting non-terminal is expanded to derive the given input string.

Bottom-Up Parsers

These are the parsers that constructs the parse tree from the leaves to the root for the given input string. In simpler terms, the construction of the parse tree is done from the bottom to the top of the tree. Here, the input string is reduced to starting non-terminal.

The act of checking whether a grammar "accepts" an input text as valid is called parsing. Parsing a given text means determining the exact correspondence between the text and the rules of a given grammar.

Self-Assessment Questions - 3

9. A data structure containing a record for each identifier, with a field for the attribute of identifier, is called "_____".
10. Symbol tables having only one scope and only "global" variables are called "_____".
11. A program which receives valid tokens and checks them against the grammar and produces valid parse trees otherwise generate syntactical errors are called _____ .
12. The graphical representation for derivations which filters out the choice regarding the replacement, order is called _____.
13. Parsers that constructsthe parse tree from the root to the leaves in preorder for the given input string, is called _____ .

14. CONTROL STATEMENTS

The control statements of a programming language are the collection of language features that govern the sequencing of control through a program.

The control statements of a Programming Language consist of constructs for control transfer, conditional execution, iteration control, and procedure calls.

Control transfer, conditional execution, and iterative constructs

Control transfers implemented through conditional and unconditional goto's are the most primitive control structure. When the target language of a compiler is a machine language, the compilation of control transfers is analogous to the assembly of forward or backward goto's in an assembly language program. Hence similar techniques based on the use of a label table can be used.

Control structures like *if, for, or while cause* a significant semantic gap between the Programming Language (PL) domains and the execution domain because the control transfers are implicit rather than explicit. This semantic gap is bridged in two steps. In the first step, a control structure is mapped into an equivalent program containing explicit goto. Since the destination of a go-to may not have a label in the source program; the compiler generates its labels and puts them against the appropriate statements. Figure 7.9 illustrates programs equivalent to the if and while statements the labels int1, int2 are introduced by the compiler for its purposes. In the second step, these programs are translated into assembly programs.

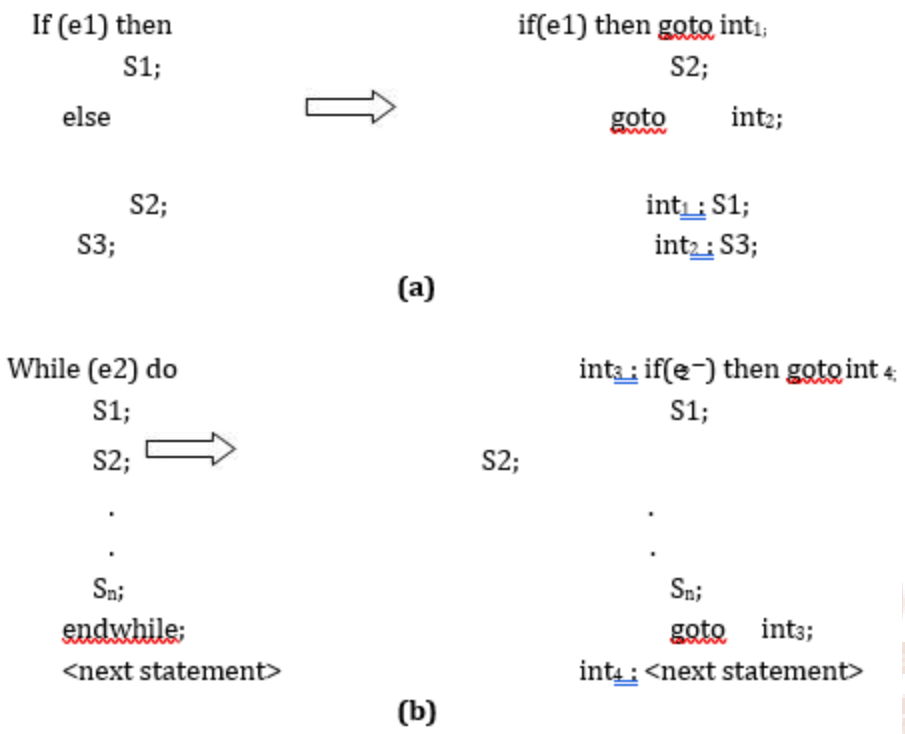


Figure 7.9: Control structure Interpretation

Function and Procedure calls

A function call, viz., the call on fn_1 in the statement

X: = fn_1(y, z) +b*c;

Executes the body of fn1, and returns its value to the calling program. In addition, the function call may also result in some side effects.

A *side effect* of a function call is a change in the value of a variable that is not local to the called function.

A procedure call only achieves a side effect which, it doesn't return a value. Since all considerations except the return of a value are analogous to function and procedure calls; in the following only compilation of function calls are discussed.

While implementing a function call, the compiler must ensure the following:

1. Actual parameters are accessible in the called function
2. The called function can produce side effects according to the rules of the PL.
3. Control is transferred to and is returned from the called function.
4. The function value is returned to the calling program.
5. All other aspects of the execution of the calling programs are unaffected by the function call.

The compiler uses a set of features to implement function calls. These are described below.

1. **Parameter list:** The parameter list contains a descriptor for each actual parameter of the function call.
2. **Save area:** The called function saves the contents of CPU registers in this area before beginning its execution.
3. **Calling conventions:** These are execution time assumptions shared by the called function and its caller.

Parameter passing mechanisms

Language rules for parameter passing define the semantics of parameter usage inside a function, thereby defining the kind of side effects a function can produce on its actual parameters.

- Call by value

  In this mechanism, values of actual parameters are passed to the called function. These values are assigned to the corresponding formal parameters.

- Call by value-result

  This mechanism extends the capabilities of the call-by-value method by copying the values of formal parameters back into corresponding actual parameters at return. Thus, side effects are realized onreturn.

- Call by reference

In this mechanism, the address of an actual parameter is passed to the called function.

- Call by name

  This parameter transmission mechanism has the same effect as if every occurrence of a formal parameter in the body of the called function is replaced by the name of the corresponding actual parameter.

15. SIMPLE INTERPRETER DESIGN

In this section, let us discuss the interpreter design technique. This section discusses the components involved in the interpretation process.

The interpretation can be done in two ways. In a first way, the source code is taken, and corresponding executable codes are generated. Here there is no preprocessing done. The source program is taken as such. Such a type of interpretation is permitted in interpreter base systems. In this case, the source program is kept as such. The overhead is more in these systems.

In a second way, some preprocessing is done. The source program is converted into an intermediate representation after preprocessing. This intermediate representation is used for interpretation. The overhead is comparatively less in this type of interpretation. The speed of interpretation is increased due to the use of intermediate representation.

The interpretation process has three major components. They are listed below.

1. Micro Programs
2. Symbol Table and
3. Datastore

The input and output of an interpreter are:

Input: Source program in HLL

Output: Executable machine code

Microprograms

Consider an executable statement in the source program to be interpreted. This statement is decoded to identify its actual operation of it. The interpreter systems run on microprogrammed control units. In the microprogrammed control , control memory will be present. The control memory keeps the microprograms, which correspond to the executable statements in the source program. A ***microprogram*** is a set of micro-instructions to carrying out a computation.

After decoding the meaning of the source program statement, the corresponding microprogram will be invoked. This microprogram will generate control signals. The control signals are responsible for execution. The control signals are software controlled. This is how the source program is interpreted.

Symbol Table

The symbol table maintains the details of the symbols present in the source program. The symbols are identifier names, labels or name of any program segments. The attribute of symbols is maintained in the symbol tables. There are a lot of data structures available for symbol tables.

Data Store

The data store keeps the necessary data items for variables, identifiers, and others. Various distinct data types are present and used during the interpretation of the source program.

Self-Assessment Questions - 4

14. The collection of language features that govern the sequencing of control through a program is called "_____".
15. A change in the value of a variable that is not local to the called function is called "_____".
16. The mechanism in which the address of an actual parameter is passed to the called function is called "_____".
17. The mechanism in which values of actual parameters are passed to the called function is called "_____".

16. SUMMARY

Let us recapitulate the important points discussed in this unit:

- An Interpreter may be a program that executes the source code directly, translates the source code into some efficient intermediate representation, and immediately executes this or explicitly executes stored precompiled code made by a compiler that is part of the interpreter system.
- Compiler is a translator. It translates the program written in a high-level language into an object language, which is a low-level language such as an assembly language or machine language.
- The interpreter scans the complete program and breaks the character stream into tokens (words).
- A symbol table is a data structure containing a record for each identifier, with a field for the attribute of the identifier.
- The control statements of a Programming Language consist of constructs for control transfer, conditional execution, iteration control, and procedure calls.
- The interpretation process has three major components: 1. Micro Programs 2. Symbol Table and 3. Data storage

17. GLOSSARY

Ambiguous grammar: a grammar that allows some sentence or string to be generated or parsed with two or more distinct parse trees.

Compiler: a program that translates from one programming language to another, typically from a high-level language such as Java to machine language

Derivation: a list of steps that shows how a sentence in a language is derived from grammar by application of grammar rules.

Grammar: a formal specification of a language consisting of a set of nonterminal symbols, a set of terminal symbols or words, and production rules that specify transformations of strings containing nonterminal into other strings.

Parsing is the process of reading a source language, determining its structure, and producing intermediate code.

Three-address code: is a linearized representation of a syntax tree or a dig in which explicit names correspond to the interior nodes of the graph.

18. TERMINAL QUESTIONS

Short Answer Questions

1. What is an Interpreter? Describe the differences between a Compiler and an Interpreter.
2. Describe Compiler. Explain different phases of compilation.
3. Describe briefly scanning and parsing.
4. Explain briefly about different control statements used in Interpretation.
5. Explain the Simple Interpreter design in detail.

19. ANSWERS

A. Self-Assessment Questions

1. Language translator
2. True
3. Three Address Code
4. Tokens
5. Parse Tree
6. Just-in-time compilation
7. Lexical analysis (scanning)
8. Tokens
9. Symbol Table
10. Simple Symbol Table
11. Parser
12. Parse Tree
13. Top-down Parser
14. Control Statement
15. Side effect
16. Call by reference
17. Call by value

B. Short Answer Questions

1. An Interpreter is a program that executes other programs. An Interpreter may be a program which executes the source code directly, translates source code into some efficient intermediate representation and immediately executes this or explicitly executes stored precompiled code made by a compiler which is part of the interpreter system. Refer to sections 2 and 10 for detail.
2. Compiler is a translator. It translates the program written in a high-level language into an object language, which is a low-level language such as an assembly language or machine language. Refer to section 4 to 9 for detail.

3. During scanning, the Interpreter scans the complete program and breaks the character stream into tokens (words). A parser (also called syntax analyzer) is a program which receives valid tokens and checks them against the grammar and produces valid parse trees; otherwise generates syntactical errors. Refer to sections 11 and 13 for detail.
4. The control statements of a programming language are the collection of language features that govern the sequencing of control through a program. Refer to section 14 for detail.
5. The interpretation can be done in two ways. In a first way, the source code is taken, and corresponding executable codes are generated. In a second way, some preprocessing is done. The source program is converted into an intermediate representation after preprocessing. Refer section 15 for detail.

20. SUGGESTED BOOKS

- Dhamdhere (2002). Systems programming and operating systems Tata McGraw-Hill.
- M.Joseph (2007). System software, Firewall Media.