UNIT-5

COMPILER AND INTERPRETER

Principles of Compilers

Introduction

The word *compilation* is used to denote the task of translating *high level language* (HLL) programs into machine language programs. Though the objective of this task of translation is similar to that of an *assembler*, the problem of compilation is much more complex than that of an assembler.
A *compiler* is a program that does the compilation task. A compiler recognises programs in a particular HLL and produces equivalent output programs appropriate for some particular computer configuration (hardware and OS). Thus, an HLL program is to a great extent independent of the configuration of the machine it will eventually run on, as long as it is ensured that the program is compiled by a compiler that recognises that HLL and produces output for the required machine configuration.
It is common for a machine to have compilers that would translate programs to produce executable for that machine (*hosts*).
But there also are compilers that runs on one type of machine but the output of which are programs that shall run on some other machine configuration, such as generating an MS-DOS executable program by compiling an HLL program in UNIX. Such a compiler is called a ***cross compiler*****.**
Another kind of translator that accepts programs in HLL are known as *interpreters*.
An interpreter translates an input HLL program and *also runs the program on the same machine*. Hence the output of running an interpreter is actually the output of the program that it translates.

Important phases /Aspects of Compilation

The following is a typical breakdown of the overall task of a compiler in an approximate sequence –

1. Lexical analysis
2. Syntax analysis
3. Intermediate code generation
4. Code optimisation
5. Code generation.

Like an assembler, a compiler usually performs the above tasks by making multiple passes over the input or some intermediate representation of the same. The compilation task calls for intensive processing of information extracted from the input programs, and hence data structures for representing such information needs to be carefully selected. During the process of translation a compiler also detects certain kinds of errors in the input, and may try to take some recovery steps for these.

Lexical Analysis

- Lexical analysis in a compiler can be performed in the same way as in an assembler. Generally in an HLL there are more number of tokens to be recognised - various keywords (such as, *for, while, if, else*, etc.)
- punctuation symbols (such as, comma, semi-colon, braces, etc.)
- operators (such as arithmatic operators, logical operators, etc.)
- identifiers, etc
- Tools like *lex* or *flex* are used to create lexcical analysers.

Syntax Analysis

Syntax analysis deals with recognising the structure of input programs according to known set of *syntax rules* defined for the HLL. This is the most important aspect in which HLLs are significantly different from lower level languages such as assembly language.
In assembly languages the syntax rules are simple which roughly requires that a program should be a sequence of statements, and each statement should essentially contain a mnemonic followed by zero or more operands depending on the mnemonic.
Optionally, there can be also be an identifier preceding the mnemonic. In case of HLLs, the syntax rules are much more complicated.
In most HLLs the notion of a statement itself is very flexible, and often allows *recursion*, making nested constructs valid.
. One commonly used formalism for this purpose is the *Context Free Grammar (CFG)*. CFG is a formalism that is more powerful than *regular grammars* (used to write regular expressions to describe *tokens* in a lexical analyser).
Recursion, which is a common feature in most constructs of HLLs, can be defined using a CFG in a concise way, whereas a regular grammar is incapable of doing so.

During syntax analysis, the compiler tries to apply the rules of the grammar of the input HLL given using BNF, to recognise the structure of the input program. This is called *parsing* and the module that performs this task is called a *parser*.

Intermediate Code Generation

Having recognised a given input program as valid, a compiler tries to create the equivalent program in the language of the target environment. In case of an assembler this translation was somewhat simpler since the operation implied by the mnemonic opcode in each statement in the input program, there is some equivalent machine opcode. The number of operands applicable for each operation in the machine language is the same as allowed for the corresponding assembly language mnemonic opcodes. Thus for the assembly language the translation for each statement can be done for each statement almost independently of the rest of the program. But, in case of an HLL, it is futile to try to associate a single machine opcode for each statement of the input language. One of the reasons for this is, as stated above, the extent of a statement is not always fixed and may contain recursion. Moreover, data references in HLL programs can assume significant levels of abstractions in comparisons to what the target execution environment may directly support. The task of associating meanings (in terms of primitive operations that can be supported by a machine) to programs or segments of a program is called *semantic processing*.

<u>Syntax Directed Translation</u>

Though it is not entirely straightforward to associate target language operations to statements in the HLL programs, the CFG for the HLL allows one to associate *semantic actions* (or implications) for the various syntactic rules. Hence in the broad task of translation, when the input program is parsed, a compiler also tries to perform certain semantic actions corresponding to the various syntactic rules that are eventually applied. However, most HLLs contain certain syntactic features for which the semantic actions are to be determined using some additional information, such as the contents of the symbol table. Hence, building and usage of data-structures such as the symbol table are an important part of the semantic action that are performed by the compiler.

Upon carrying out the semantic processing a more manageable equivalent form of the input program is obtained. This is stored (represented) using some *Intermediate code* representation that makes further processing easy. In this representation, the compiler often has to introduce several temporary variables to store intermediate results of various operations. The language used for the intermediate code is generally not any particular machine language, but is such which can be efficiently converted to a required machine language (some form of assembly language can be considered for such use).

Code Optimisation

The programs represented in the intermediate code form usually contains much scope for optimisation both in terms of storage space as well as run time efficiency of the intended output program. Sometimes the input program itself contains such scope. Besides that, the process of generating the intermediate code representation usually leaves much room for such optimisation. Hence, compilers usually implement explicit steps to optimise the intermediate code.

Code Generation

Finally, the compiler converts the (optimised) program in the intermediate code representation to the required machine language. It needs to be noted that if the program being translated by the compiler actually has dependencies on some external modules, then *linking* has to be performed to the output of the compiler. These activities are independent of whether the input program was in HLL or assembly language.

Run-time Storage Administration

One of the important aspects of the semantic actions of a compiler is to ensure an efficient and error free run-time storage model to the output program. Most modern programming languages allow some extent of *block-structuring*, nesting of constructs, and recursion of subroutines. All these calls for an efficient modelling of data storage that is dynamic, and it turns out that a *stack* meets much of the criteria. Thus allocation and access of storage for program variables, subroutine parameters, and compiler generated internal variables on the stack is an important part of the task of a compiler.

Memory allocation

Memory allocation definition

Memory allocation is the process of reserving virtual or physical computer space for a specific purpose (e.g., for computer programs and services to run). Memory allocation is part of the management of computer memory resources, known as memory management. Through memory allocation, computer programs and services are assigned a specific memory portion, depending on how much memory they need.

How memory allocation works

- Memory allocation is a computer hardware operation managed through the operating system.
- Through the process of memory allocation, a portion of the computer memory is set aside for running programs and processes. How much memory is allocated depends on the program's requirements.
- When the program or application has finished its operation, the memory is released and assigned to another program.

Types of memory allocation

- **Static memory allocation (compile-time).** Programs and services are allocated memory at compile time. Compile time refers to the period when the programming code is converted to machine code for the computer to understand.
- **Dynamic memory allocation (run-time).** Programs and services are allocated memory at run time or execution time. Dynamic memory allocation allows computers to use memory space more efficiently.

Compilation of Expressions

In this section:

- Compiling Expressions Using the DEFINES Parameter
- Compiling Expressions Using the COMPUTE Parameter

Compiling Expressions Using the DEFINES Parameter

Two expression compilers are available on OS. By issuing the appropriate command, you can select one of them or disable compilation of expressions. Both compilers cannot be active for the same request:

- The DEFINE compiler compiles only those expressions that are found in DEFINE fields referenced in TABLE requests, but it provides much faster execution of those expressions than the other compiler. It compiles expressions using the arithmetic operations built into the underlying operating system, and is, therefore, referred to as the native compiler. Compilation takes place at TABLE run time.

  This compiler is invoked by issuing the command
  SET DEFINES = COMPILED.

- The other compiler is invoked with the command SET COMPUTE = NEW. This compiler provides expression compilation for DEFINE, IF, and WHERE commands in TABLE procedures. Under this compiler, expressions are compiled at DEFINE time. Therefore, compilation may be invoked for expressions that are never actually used in a request.

Among the benefits of the DEFINE compiler are:

- Compilation of only those expressions that are actually used in the TABLE request.
- Much faster execution of expressions containing complex calculations on long packed fields.
- Compilation of date expressions.

After the native DEFINE compiler is invoked, any request that uses a DEFINE expression causes the expression to be compiled and then loaded into the system. For each record of the request that needs computation, the system executes the generated code. This compiler is most effective with TABLE requests that include a large number of DEFINE fields and read a large number of records because the speed of evaluation per record in such requests offsets the extra compilation and load steps.

Syntax: How to Compile DEFINE Expressions

SET DEFINES = {COMPILED|OLD}

where:

COMPILED

> Implements expression compilation at request run time, compiling only those DEFINEs that are used in the request.

OLD

> Leaves expression compilation up to the control of the current SET COMPUTE value. OLD is the default value. If you issue the SET DEFINES = OLD command, the COMPUTE parameter is automatically set to NEW.

Syntax: How to Control Expression Compilation Using the COMPUTE Parameter

```
SET COMPUTE = {NEW|OLD|NATV}
```

where:

NEW

Compiles DEFINE calculations when a request is executed.

OLD

Does not compile DEFINE calculations when a request is executed. The old logic is used.

NATV

Compiles DEFINE calculations using the native compiler. This setting is also activated by the SET DEFINES=COMPILED command, which is the default setting.

Interaction Between SET DEFINES and SET COMPUTE

Two expression compilers are available, but only one can be activated for any request. Activating either compiler automatically deactivates the other compiler:

- Issuing the SET DEFINES=COMPILED command activates the new compiler and deactivates the old compiler by automatically setting the value of the COMPUTE parameter to NATV (native compiler).
- Issuing the SET DEFINES=OLD command deactivates the new compiler and automatically activates the old compiler by automatically setting the value of the COMPUTE parameter to NEW .
- Issuing the SET COMPUTE command with either the OLD or NEW setting deactivates the new compiler. The OLD setting also deactivates the old compiler.

Therefore, you can select either compiler by issuing the SET DEFINES command. DEFINES=COMPILED selects the new compiler, DEFINES=OLD selects the old compiler. To turn compilation off, issue SET COMPUTE=OLD.

The new compiler is recommended for TABLE requests that include a large number of DEFINE fields (especially those that use packed arithmetic or date expressions) and read a large number of records.

If a TABLE request retrieves a large number of records or if the DEFINE fields use packed arithmetic (especially with long packed fields) or date expressions, the new compiler is likely to provide the most benefit.

Compilation of control structure

We have to know about control structure

Control structures: These are fundamental building blocks in programming that dictate the flow of code execution depending on certain conditions or predefined logic.

These structures help you organise your code effectively and optimise its performance.

Three Control Structures in Programming: Sequence, Selection, and Iteration

A detailed understanding of each of the three main types of control structures is essential for effective programming

- **Sequence**: This control structure represents the linear and sequential execution of code, where statements are executed one after another in the order they appear. This is the simplest type of programming control structure and forms the foundation of most programs.

- **Selection**: This control structure enables a program to choose between two or more paths of execution based on specific conditions. Selection structures include conditional statements such as if-else and switch-case, which use Boolean expressions to evaluate the conditions and execute the appropriate block of code.
- **Iteration**: This control structure allows certain blocks of code to be executed repeatedly as long as a condition remains true. Iteration structures include loops like the for loop, while loop, and do-while loop.

Compilation of Sequence Control Structures

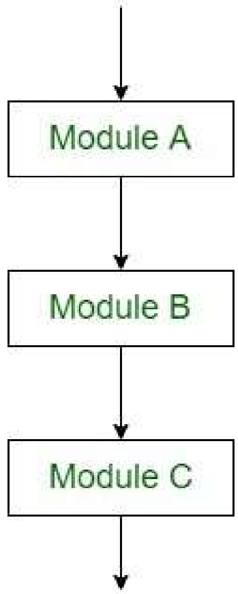
The sequence control structure is employed in various scenarios where statements need to be executed in a linear fashion. Here are some examples of how the sequence control structure can be used:

- Computing the sum of a series of numbers
- Reading input data from a file or user
- Writing output data to a file or console
- Performing mathematical calculations

While sequence structures might appear simple, they serve as the backbone of programs that rely on other types of control structures, such as selection and iteration.

1. **Sequential Logic (Sequential Flow)**

Sequential logic as the name suggests follows a serial or sequential flow in which the flow depends on the series of instructions given to the computer. Unless new instructions are given, the modules are executed in the obvious sequence. The sequences may be given, by means of numbered steps explicitly. Also, implicitly follows the order in which modules are written. Most of the processing, even some complex problems, will generally follow this elementary flow pattern.



Sequential Control flow

2. **Selection Logic (Conditional Flow)**
   Selection Logic simply involves a number of conditions or parameters which decides one out of several written modules. The structures which use these type of logic are known as **Conditional Structures**. These structures can be of three types:
   Single Alternative : This structure has the form:

   If (condition) then:

   [Module A]

   [End of If structure]

   Double Alternative: This structure has the form:

   If (Condition), then:

   [Module A]

Else:

[Module B]

[End if structure]

Multiple Alternatives: This structure has the form:

If (condition A), then:

[Module A]

Else if (condition B), then:

[Module B]

..

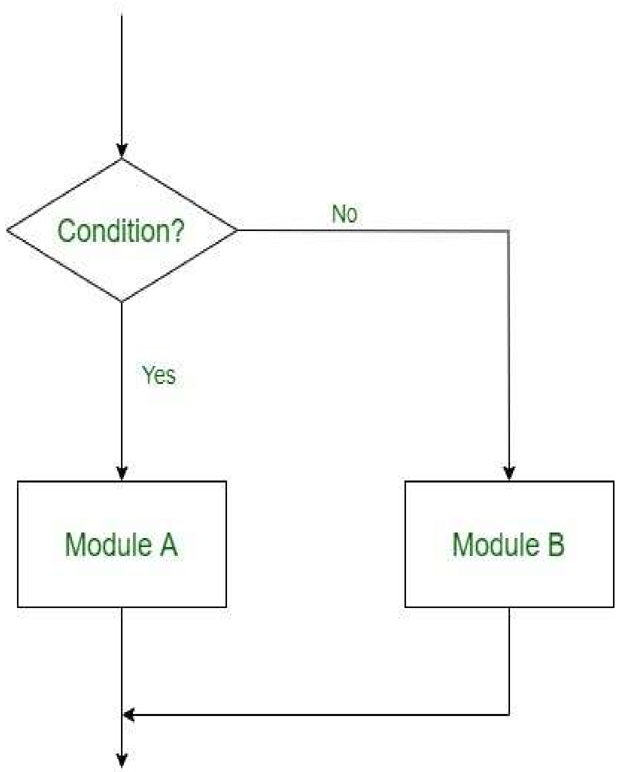
..

Else if (condition N), then:

[Module N]

[End If structure]

In this way, the flow of the program depends on the set of conditions that are written. This can be more understood by the following flow charts:



3. **Iteration Logic (Repetitive Flow)**
   The Iteration logic employs a loop which involves a repeat statement followed by a module known as the body of a loop.
   The two types of these structures are:
   - **Repeat-For Structure**: This structure has the form:

   Repeat for i = A to N by I:

   [Module]

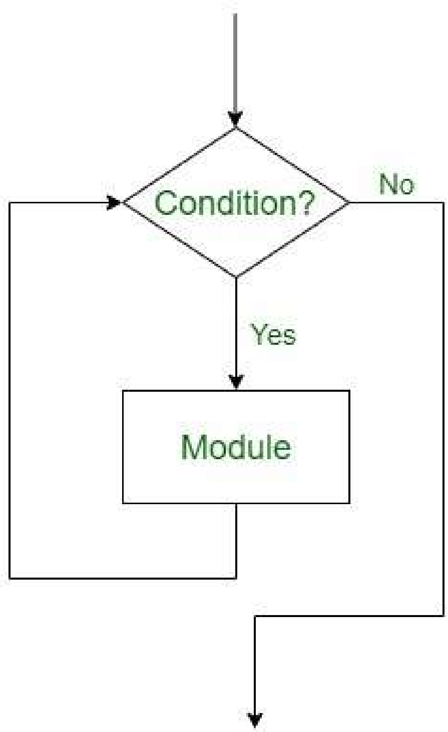
   [End of loop]

   Here, A is the initial value, N is the end value and I is the increment. The loop ends when A>B. K increases or decreases according to the positive and negative value of I respectively.

Repeat-While Structure
It also uses a condition to control the loop. This structure has the form:
Repeat while condition:

[Module]

[End of Loop]



Repeat While Flow

Software tools for program development

What is Software Tool?

A software tool is a system program that interfaces a program with the entity generating its input data or interfaces the results of a program with the entity consuming them.

- It is a set of computer programs used by developers to create, maintain, debug, or support other applications and programs.
- It can be code editors, language libraries, debuggers, performance analysis tools, GUI designers, etc.
- There are some factors that need to be considered before selecting a software tool like the usefulness of the tool, company criteria, integration of one tool with another, etc.
- Software tools assist in all the activities of the software lifecycle.
- These are used to accomplish and investigate the business processes, document the development process, and optimize all the processes.
- These help developers easily maintain the workflow of the project.

Common Software Development Tools Types

Software tools are programs or applications that assist in the development, testing, and maintenance of software. Some common software development tools include:

1. **Integrated Development Environments (IDEs):** IDEs are specialized software tools that provide a complete environment for software development, including a text editor, compiler, debugger, and other tools.
2. **Source Code Management (SCM) tools:** SCM tools allow developers to manage and track changes to their source code, as well as collaborate with other team members.
3. **Debugging tools:** Debugging tools allow developers to identify and fix errors in their code.
4. **Test automation tools:** Test automation tools allow developers to automate the testing process, making it more efficient and accurate.
5. **Project management tools:** Project management tools help developers to plan, track, and manage their software development projects.
6. **Collaboration tools:** Collaboration tools allow developers to communicate and work with other team members remotely.

7. **Static code analysis tools:** Static code analysis tools analyze source code for potential issues, such as security vulnerabilities or coding errors.
8. **Performance analysis tools:** Performance analysis tools help developers to identify and optimize the performance of their software.
9. **Documentation tools:** Documentation tools help developers to create and maintain documentation for their software projects.
10. **Configuration management tools:** Configuration management tools help developers manage and track software configuration changes.

Best Software Development Tools

Below are some of the software development tools:

1. Git: A version control system that allows developers to track changes to their code and collaborate with other team members. Features include branching, merging, and merging conflicts.

2. IntelliJ IDEA: A popular integrated development environment (IDE) for Java development, with support for a wide range of programming languages and frameworks. Features include code completion, refactoring, and debugging tools.

3. Eclipse: Another popular IDE, with a focus on Java development but support for a range of other languages as well. Features include code completion, refactoring, and debugging tools.

4. Visual Studio: A comprehensive IDE from Microsoft, with support for a wide range of programming languages and platforms. Features include code completion, refactoring, debugging, and integration with other Microsoft tools.

5. PyCharm: An IDE specifically designed for Python development, with support for scientific computing and data science tools. Features include code completion, debugging, and integration with version control systems.

6. Xcode: The IDE for Apple's platforms, including macOS, iOS, iPadOS, watchOS, and tvOS. Features include code completion, debugging, and integration with Apple's development tools.

7. Android Studio: The official IDE for Android development, with support for building, testing, and debugging Android apps. Features include a visual layout editor, emulator, and integration with Google's developer tools.

8. Node.js: A JavaScript runtime built on Chrome's V8 JavaScript engine, allowing developers to run JavaScript on the server side. Features include support for asynchronous programming and a large ecosystem of open-source libraries.

9. npm: The default package manager for the Node.js runtime, allowing developers to easily install and manage third-party libraries and dependencies. Features include version management and automatic dependency resolution.

10. Gradle: A build automation tool for Java and other programming languages, allowing developers to automate the process of building, testing, and deploying their code. Features include support for multiple languages and platforms and integration with a range of tools.

Editors in computer technology

In the field of programming, the term editor usually refers to source code editors that include many special features for writing and editing code. Notepad, Wordpad are some of the common editors used on Windows OS and vi, emacs, Jed, pico are the editors on UNIX OS. Features normally associated with text editors are — moving the cursor, deleting, replacing, pasting, finding, finding and replacing, saving etc.

Types of Editors
There are generally five types of editors as described below:

1. **Line editor**: In this, you can only edit one line at a time or an integral number of lines. You cannot have a free-flowing sequence of characters. It will take care of only one line.
   Ex : Teleprinter, edlin, teco

2. **Stream editors**: In this type of editors, the file is treated as continuous flow or sequence of characters instead of line numbers, which means here you can type paragraphs.
   Ex : Sed editor in UNIX
3. **Screen editors**: In this type of editors, the user is able to see the cursor on the screen and can make a copy, cut, paste operation easily. It is very easy to use mouse pointer.
   Ex : vi, emacs, Notepad
4. **Word Processor**: Overcoming the limitations of screen editors, it allows one to use some format to insert images, files, videos, use font, size, style features. It majorly focuses on Natural language.

5. **Structure Editor**: Structure editor focuses on programming languages. It provides features to write and edit source code.
   Ex : Netbeans IDE, gEdit.

Debug Monitor

A debug monitor, simply put, is a tool that helps to find and reduce the number of bugs and defects in a computer program or any electrical device within or attached to the computer in order to make it act the way it should.

Debug monitors give debugging support for a program. A debug monitor executes the program being debugged in its own control thereby giving execution efficiency throughout debugging. There are debug monitors which are language independent and can handle programs written in several languages.

Debug monitor give the following facilities.

It provides information about the program's execution, such as memory usage, variable values, and program flow, to help developers identify and fix errors or bugs in the code.

- Setting breakpoints into the program
- Initiating a debug conversation while control reaches a breakpoint.
- Displaying variable's values
- Assigning new values to variables.
- Testing in defined assertions and predicates including program variables.

USER INTERFACE

A software system's user interface is how a user experiences it for the first time. Any software system must therefore fulfill user requirements. UI primarily carries out two tasks:

- Accepting user input.
- Showing the results.

User Interface (UI) defines the way humans interact with the information systems. In Layman's term, User Interface (UI) is a series of pages, screens, buttons, forms and other visual elements that are used to interact with the device.

Every app and every website has a user interface. User Interface (UI) Design is the creation of graphics, illustrations, and use of photographic artwork and typography to enhance the display and layout of a digital product within its various device views. Interface elements consist of input controls (buttons, drop-down menus, data fields), navigational components (search fields, slider, icons, tags), informational components (progress bars, notifications, message boxes).

Types of User Interface (UI)

- **GUI (Graphical User Interface):** Visual user interface output (keyboard and monitor) with a tactile user interface input.
- **Menu Driven Interface:** An UI that uses a menu of options to navigate a program or website is known as a menu-driven UI. For instance, ATMs have user interfaces that are menu-driven and simple to use.
- **Form Based Interface**: Form-based user interfaces provide a small number of options for users to choose from when entering data into a program or application. For instance, a device's settings menu is form-based.
- **Touch user interface:** Haptic or tactile user interface. Haptic input is used by most smartphones, tablets, and other devices with touch screens.
- **Voice user interface:** Auditory commands are used to communicate between humans and machines. GPS, talk-to-text gadgets, and virtual assistants are a few examples.

Importance of User Interface (UI)

- How you present your product matters the most. The presentation (the interface) of a badly designed application or website can drive away the incoming users and leave a bad impression on them. Navigation through a site can be made efficient and simple by effective UI design.
- Great design is great business The color scheme, layout, graphics, tab and button placement, typography usage and other design elements determine how well the site/app communicates visually with the user.
- At a given time, finding right thing at right place is essential Consistent navigation that is equally easy to locate and browse through is basic need of any app/website. If navigation is complex for a user to work, there is no way he would want to work on that site/app.

Advantages of User Interface (UI)

Some advantages of User Interface are mentioned below:

- No need to learn complex commands/languages for working with UI.
- Easiness for non-technical people. A beginner can navigate through a site with ease if its simple and well informative.
- Usage of blocks and typography makes user experience better.

- Easy setup and ready to start working are awesome. Hiding the complexity of actions from the user and display only the required information is key to good interface.

Disadvantages of User Interface(UI)

Some disadvantages of User Interface are listed below:

- When not properly built, it can be very difficult to work with.

- Takes time to built a Perfect UI.