Chapter 1 :Introduction to Different Programming Paradigms

Learning Objectives

At the end of this module, you will be able to:

- Define programs and its components.
- Explain various programming paradigms and languages.
- Describe classification of languages.
- Describe role of programming languages.

Introduction

Since 1950s, computer science has been an exciting field. It is now used in every aspect of life that includes medical science, telecommunication, space exploration, modelling, computer-aided design, automated navigation, manufacturing, surgery, presentation tools, commerce, etc. Computers are now embedded in various gadgets, such as cell phones, automobiles, spacecraft, high-end ovens, home security systems, etc. All these activities can be performed with the help of programmers, who have solved complex programs and presented them in high-level instructions. These instructions are translated into low-level machine instructions by automated language instructors. The high-level language and language constructs are used to express the solutions for the complex problems.

In this chapter, you will learn about programs and their components. You will also learn about the history of programming paradigms and languages, and the classification of languages and role of programming languages.

1.1 Introduction to Programs and its Components

In order to automate a process or solve a problem, a system analyst creates a model of system, finds input and output behaviour, and then connects the modules with the help of a flow chart. These programs are the specifications that are used to process the data and handle data flow between modules. The solution of a problem is described at a high level to make it easily understandable to humans. However, computer only understands low-level machine instructions. Therefore, a translation process is needed to map the high-level instructions to low-level instructions.

A program is a sequence of meaningful symbols to specify the solution of a complex problem as shown in Figure 1:

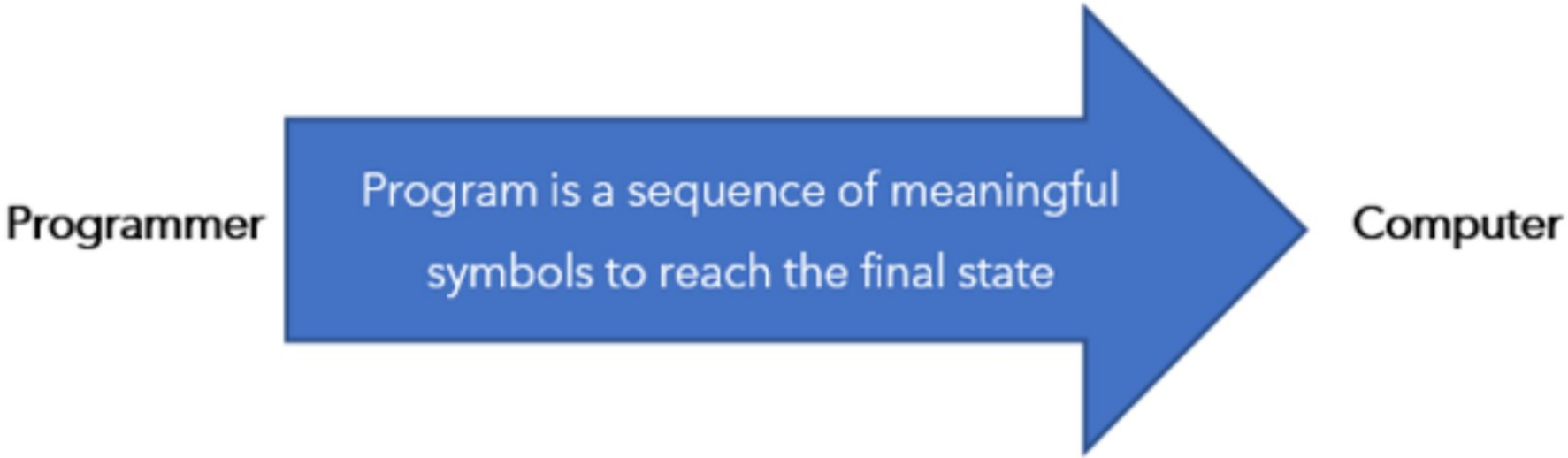


Figure 1: Definition of Programs

A program consists of three components — logic, abstraction and control. Logic is the high-level specification of the solution to a problem. It needs breaking up a complex problem into simple problems and then combining the solutions of simple problems to create a final solution. Abstraction is defined as modelling an entity with the desired

Figure 1: Definition of Programs

A program consists of three components — logic, abstraction and control. Logic is the high-level specification of the solution to a problem. It needs breaking up a complex problem into simple problems and then combining the solutions of simple problems to create a final solution. Abstraction is defined as modelling an entity with the desired attributes required to solve the problem. It helps in maintaining and modifying programs more easily. Control is defined as mapping the solution of the problem.

In the context of the computers and computing, “Creating a program by following the rules of a programming language is called programming”. A program is a set of instructions or statements used to perform a specific task. To create a program for a computer, a user needs to know a programming language. There are several programming languages, such as C, C++, C#, Java and Visual Basic.

Each programming language is unique and has different syntax. However, there are some common and important concepts in all programming languages, which are the basics of any programming. The major concepts are as follows:

- Every programming language and its program should follow a sequence of commands, which means placing the right commands in the correct order. An incorrect sequence might not give the desired output. Therefore, it is necessary to use the correct sequence of steps according to the task.
- Usage of conditional structures, which means doing certain things based on yes or no, or based on true or false, or other decisions in the programs. It generates a particular output or sequence of events if some statements/instructions are true or a different output if some statements/instructions are false. Almost all programming languages use the “if…then…else” structure for fulfilling this point.
- Usage of looping structure, which means creating a list of instructions for doing something more than once. For this, the programmer needs to use some condition to run statements/instructions on a predetermined number. For example, “Do the following exercise 10 times”, where one needs to repeat a task ten times.

1.2 Abstraction in Programs

There are two types of abstractions a program can use — data abstraction and control abstraction. Data abstraction refers to displaying only the required information to the outside world without showing the background details, i.e. showing only the essential features of the application and hiding the details.

For example, a database system hides the process of storing, creating and maintaining the data. Similarly, different methods are provided by C++ classes to the outside world; however, the internal details regarding the methods and data involved are not disclosed to them.

Let's take another real-life example to explain data abstraction. Consider a television, which you can switch on and off, change the channels, adjust the volume, colour and brightness, and add external components, such as speakers and DVD players. However, the internal working principle of the TV is not known to you. You have no idea about how signals are received over the air or through a cable, how they are translated and how the images are displayed on the screen.

Control abstraction is used to classify the program statements into different groups according to their properties. For example, assertions, block of statements, selection constructs, indefinite iteration, definite iteration, functional call, procedural call, etc. The complicated implementation of functionalities when written under a subroutine with a name helps to implement control abstraction. Consider a program that needs to find whether a given number is prime so that it can be used for creating cipher keys used in encryption algorithms. Then the complex job of resolving whether the number is prime or not can be written inside a function, such as isPrime. Every time a number is to be verified for prime, it can be sent to this function. The programmer need not know how the test is performed, i.e. the implementation is hidden under the name isPrime. Thus, control abstraction can be implemented using subroutine and calls.

1.2.1 Program Execution

Some of the major criteria of program execution are:

- The language constructs of the program should be well defined.

1.2.1 Program Execution

Some of the major criteria of program execution are:

- The language constructs of the program should be well defined.
- There should be a unique meaning for the language construct as computers do not handle ambiguities.
- Every high-level instruction should translate into a low-level instruction that should consistently perform the same action.
- A computer should consistently execute the same sequence of instructions to produce the same result.

Following are the three ways to translate a high-level instruction to a low-level instruction:

1. Compile the high-level instruction before executing the program.
2. Interpret the high-level program and execute it using an implemented abstraction.
3. Compile high-level program just in time to give an effect of partially compiled code and partially interpreted code.

As shown in Figure 2, the compiler translates the high-level instruction to low-level instruction before execution. However, the interpreter translates and executes one transaction at a time:

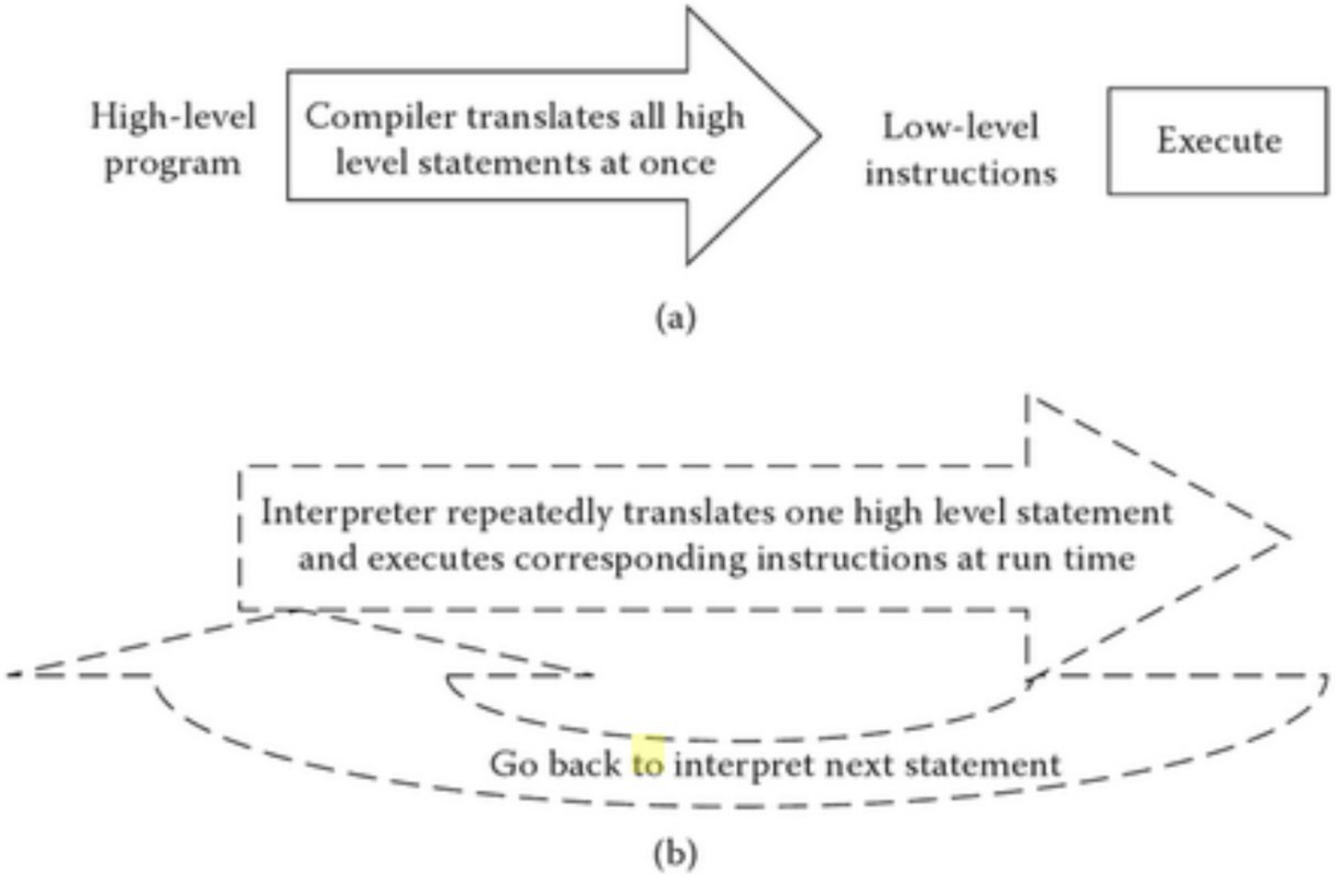


Figure 2: Compilation and Interpretation

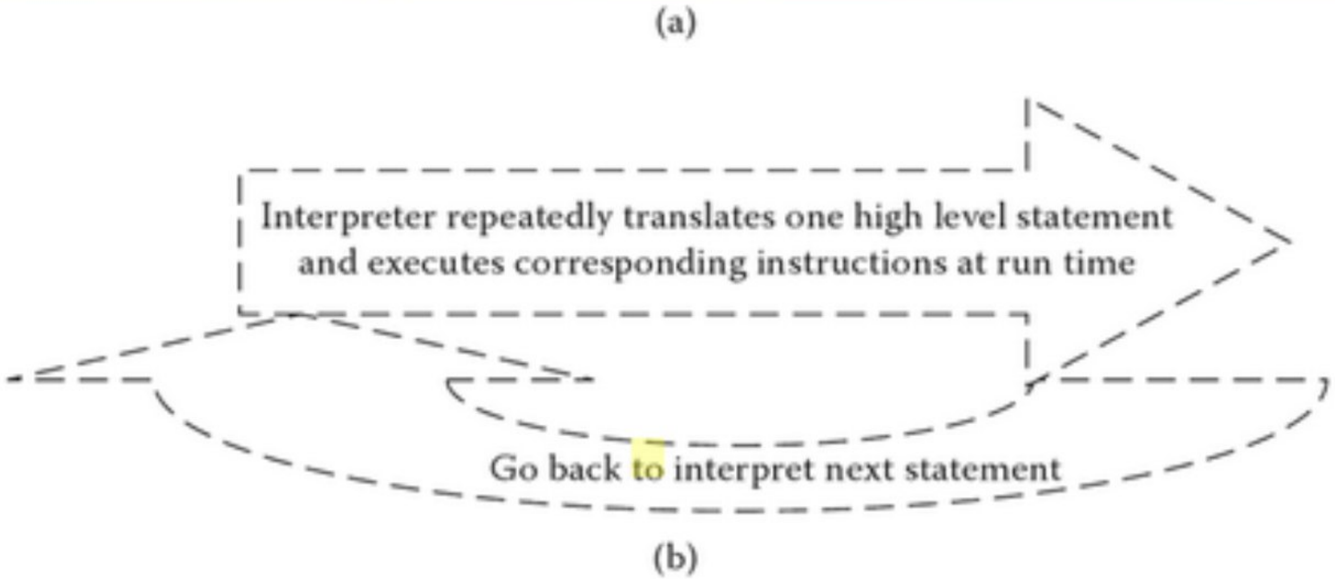


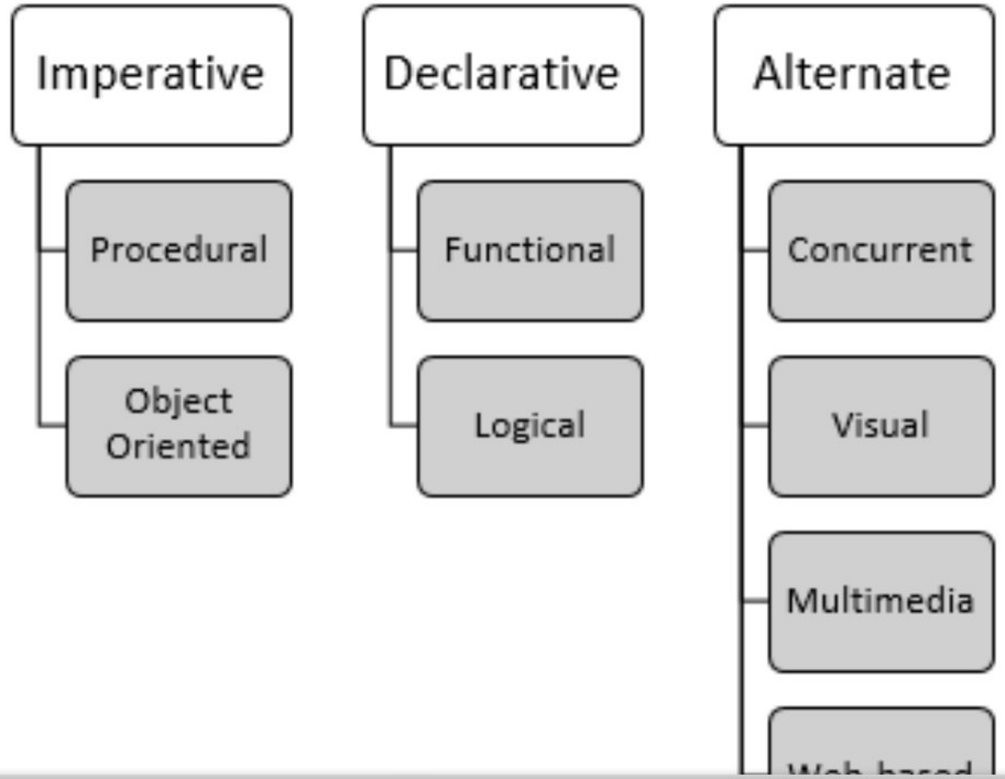
Figure 2: Compilation and Interpretation

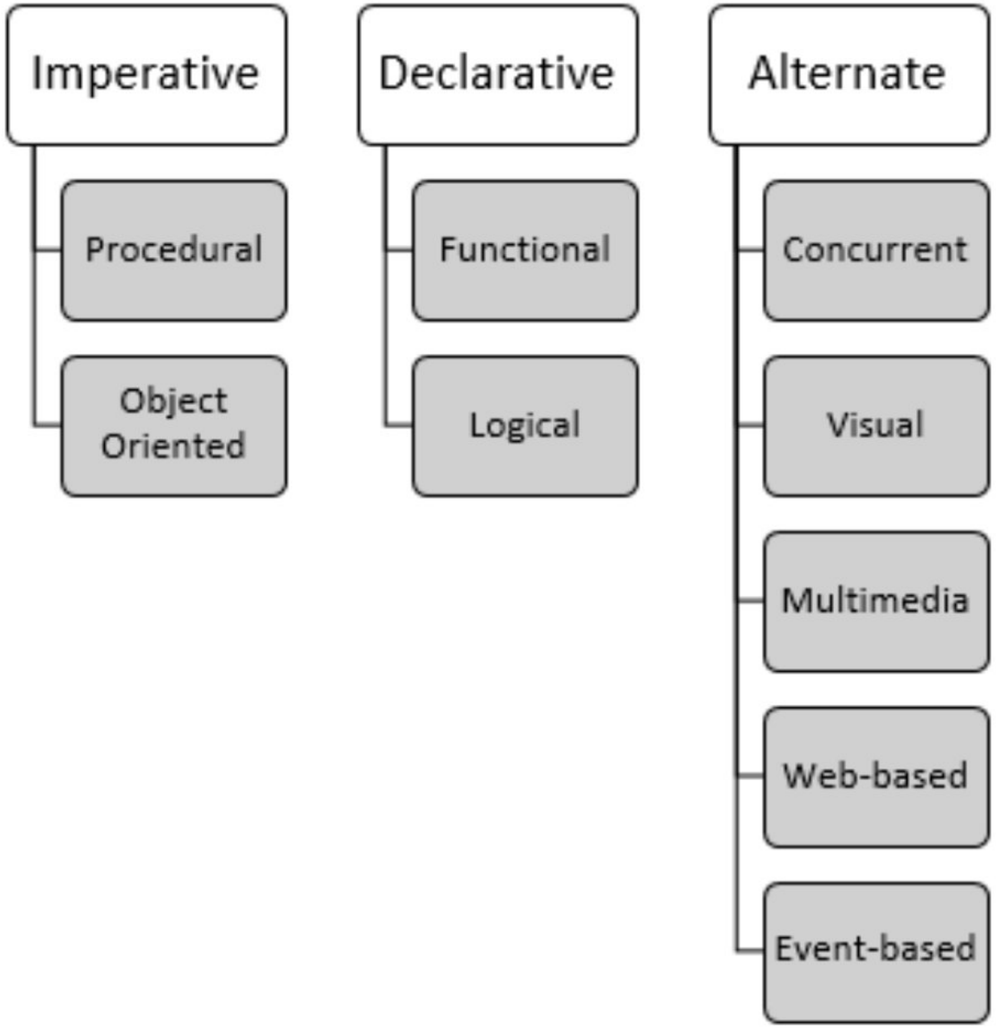
The advantages of compiled code are that there is no overhead of translation at a run time and during compilation more programming bugs are detected.

The drawback of interpreted code is that its efficiency is a lot slower than that of compiled code. Also, in interpreters, the errors cannot be detected during the program execution. These programs may crash after executing various instructions. However, they are easy to develop and were used earlier because compilers were not well developed.

1.3 History of Programming Paradigms and Languages

Programming paradigm is defined as the style of programming. With time, new programming styles have evolved. Now, most of the modern programming languages are a combination of two or more programming paradigms. The programs can be classified into one or more programming paradigms as mentioned below. Most widely noted paradigms are imperative and declarative paradigms. As newer languages came into picture, many more paradigms have come into existence and are listed under alternative programming paradigms.





- **Imperative Programming Paradigm**: In imperative programming, an assertion or assignment statement is used to change the state of the low-level von Neumann machine. Imperative programming paradigm thus focuses on "How" to get the result. Hence, in essence it can also be referred to as the **Procedural programming**

- **Imperative Programming Paradigm**: In imperative programming, an assertion or assignment statement is used to change the state of the low-level von Neumann machine. Imperative programming paradigm thus focuses on “How” to get the result. Hence, in essence it can also be referred to as the **Procedural programming paradigm**. A programmer manually translates the logic to explicitly tell the computer what to do. A variable in imperative program is mapped into a memory location and it is repeatedly modified by using assignment statements. The advantage of assignment statement is memory reuse as it is used again for storing multiple values. However, some of the disadvantages of writing into memory location multiple times are as follows:
    - If a memory location is rewritten, then the old value is lost and it makes it impossible to use the old value in the future.
    - If a memory space is allocated to another function, even then the old value is lost. If that function is unaware of the modification, then the update has corrupted the memory location. The outcome can generate incorrect solutions.
    - The interleaving of the assignment statement with logic of the program can also cause problem in the program comprehension.

Imperative programming was one of the first to be developed in the late 1950s and 1960s. Some of the popular imperative programming languages are FORTRAN, ALGOL and COBOL, and C.

- **Declarative Programming Paradigm**: In declarative programming, control is taken care by an abstract machine. A declarative program consists of logic and abstraction at the programmer level. A variable in the declarative program is used to store the value. After assigning a value to the variable, it cannot be changed. The advantages of this property are as follows:
    - There is a less possibility of side effects.
    - The old value of the variables can be restored.

The disadvantages of the write-once property are as follows:

- The memory location cannot be reused if it is not needed.
- Recursive programming is used and it has both memory and execution overheads.
- Global mutable variables that can be used by other parts of the program to improve the execution efficiency are not allowed. However, some programming languages have tried to overcome this limitation.

There are two types of declarative programming language — programming languages and programming languages. Functional programming languages are based on mathematical functions and logic programming languages are based on the predicate logic – Boolean logic, i.e. based on facts and rules.

Lisp is a programming language that combines functional programming with some imperative constructs. Prolog is a logic programming language that was used for automated theorem proving and artificial intelligence programming. After some time, other functional languages were developed that were influenced by Lisp. Some of them are Scheme, ML, Miranda, and Haskell. Now, various multiparadigm languages, such as Ruby and Scala, support the functional programming paradigm.

- **Object-oriented Programming Paradigm**: Around 1980s, object-oriented programming caught on with the software development community. With the increasing size of the program, people started understanding about modularity and software reuse to develop even more complex software. Modularity means dividing large software programs into a set of interconnected modules. In each module, their functionality does not overlap with the functionality of other modules. The advantage of modular development is that the modification in one module does not adversely affect the functionality of other modules.

To provide modularity, the objects and dates were discovered. An object has both data abstractions and methods-related functions. The information can be private, public, or protected. The notion of software reuse has been facilitated by the use of inheritance, where a subclass can inherit the methods from the parent class.

The first language to use object-oriented programming was SIMULA. After then many object-oriented programming languages have been developed. For example, C++ is an integration of the language C and the object-oriented paradigm. Many variations of Prolog have a library for object-oriented programming. Modern-day scripting languages, such as Python, Ruby, and PHP, have integrated object-oriented programming. To implement object-oriented paradigm, most of the time the variable values are altered in the program, i.e. programs show side effects which is a property of imperative paradigm. Hence, it is said to be a subtype of imperative paradigm.

- **Concurrent Programming Paradigm**: During 1980, software was bigger and hardware technology was also changing. Various fast processors were becoming available in a single computer. With this, subtasks could be mapped on the separate processors to improve the efficiency of the programs. Then, the development took two directions:
    - Parallelising compilers that can use a sequential program as input.
    - Including high-level program constructs, such as threads, forking, busy-wait-loop and remote procedure calls.

The concurrent counterparts of various languages were generated and given a generic name as "Concurrent X" or "Parallel X", where X was the name of an existing language

The concurrent counterparts of various languages were generated and given a generic name as “Concurrent X” or “Parallel X”, where X was the name of an existing language. Modern programming languages include concurrency constructs to improve the execution efficiency of the programs.

- **Visual Programming Paradigm**: Most programming languages are textual and they suffer from sequential one-dimensional programming. Then in the late 1980s, the visual programming paradigm took multiple directions. Some programming languages use symbolic representation for different data abstractions and control abstractions at low-level programming. However, visual programming was only limited to drag and drop to provide user-friendly interfaces and animations.

Now, visual programming has been used in languages, such as C#, for event-based programming. They use symbols corresponding to events and object attributes. However, it has remained dormant as there is no standardisation of symbols used in visual programming.

- **Multimedia Programming Paradigm:** This includes integration of multiple modes of visualisation, such as text, images, audio, video and gestures. With the development of web-based programming and comprehensive formats for audio, images and video, it became possible to include multimedia objects and video clips into programs for human visualisation and perception. There are many languages that are used to create 3D animated objects and motion to model virtual reality. Some of them are Alice and Virtual Reality Markup Language (VRML). In recent years, some other 3D modelling languages have been developed, such as X3D and Java 3D.
- **Web-based Programming Paradigm**: With the development of the Internet in early 1990, now users can share data, images, audio visuals, database and mobile-code located remote website. It has provided the ability of code and data mobility. If a remote resource does not want to share the code, then it can compute the data source and transmit the resulting data. While if a server does not want to get overloaded, then it can set the code to the client to perform the computation at the client’s end.

Now, there are many Internet-based languages, such as Java and SMIL, and web development languages, such as PHP, JavaScript, and XML. XML has become a popular language to represent databases, computations and animations over the Internet. Java uses an abstract machine called JVM. Its programs are interpreted using the JVM which is similar to an assembly language for zero-address machines.

A zero-address machine does not use memory address in the instructions. Instead, it uses stacks where the operands are placed on an evaluation stack, popped, evaluated and pushed back on the stack. It helps in embedding a language on modern everyday devices, such as microwave, smart homes and refrigerators.

- **Event-based Programming Paradigm:** Events are the situations that set up conditions that trigger some actions or computations. For example, clicking a mouse button or moving the mouse are events. The difference between input-driven programming and event-based programming is that input-driven programming asks for an input, waits for it and then takes an action based upon the input value. However, event-based programming never waits for any input from the external world. A modern-day

Now, visual programming has been used in languages, such as C#, for event-based programming. They use symbols corresponding to events and object attributes. However, it has remained dormant as there is no standardisation of symbols used in visual programming.

- **Multimedia Programming Paradigm:** This includes integration of multiple modes of visualisation, such as text, images, audio, video and gestures. With the development of web-based programming and comprehensive formats for audio, images and video, it became possible to include multimedia objects and video clips into programs for human visualisation and perception. There are many languages that are used to create 3D animated objects and motion to model virtual reality. Some of them are Alice and Virtual Reality Markup Language (VRML). In recent years, some other 3D modelling languages have been developed, such as X3D and Java 3D.
- **Web-based Programming Paradigm**: With the development of the Internet in early 1990, now users can share data, images, audio visuals, database and mobile-code located remote website. It has provided the ability of code and data mobility. If a remote resource does not want to share the code, then it can compute the data source and transmit the resulting data. While if a server does not want to get overloaded, then it can set the code to the client to perform the computation at the client's end.

Now, there are many Internet-based languages, such as Java and SMIL, and web development languages, such as PHP, JavaScript, and XML. XML has become a popular language to represent databases, computations and animations over the Internet. Java uses an abstract machine called JVM. Its programs are interpreted using the JVM which is similar to an assembly language for zero-address machines.

A zero-address machine does not use memory address in the instructions. Instead, it uses stacks where the operands are placed on an evaluation stack, popped, evaluated and pushed back on the stack. It helps in embedding a language on modern everyday devices, such as microwave, smart homes and refrigerators.

- **Event-based Programming Paradigm:** Events are the situations that set up conditions that trigger some actions or computations. For example, clicking a mouse button or moving the mouse are events. The difference between input-driven programming and event-based programming is that input-driven programming asks for an input, waits for it and then takes an action based upon the input value. However, event-based programming never waits for any input from the external world. A modern-day language, such as C#, uses event-based programming to model animation.

1.4 Classification of Languages

Languages can be classified using various criteria, such as programming paradigms, implementation models, the notions of types used in the programming languages and their intended problem domains. However, it is tough to clearly separate many modern-day languages as they include a combination of features and paradigms.

1.4.1 Programming Paradigms-based Classification

In the earlier sections, you have already learnt about various programming paradigms. For example, versions of FORTRAN, C, ALGOL 60, Pascal, ADA and PL/I were imperative languages; some versions of Lisp, ML and Scheme are popular functional programming languages; Prolog is a popular logic programming language; and C++, Java, Scala, Ruby, Python and X10 are multiple programming paradigm languages.

Now, C++ is an integration of imperative programming paradigm and object-oriented programming paradigm. Common Lisp Object-Oriented System (CLOS) is an integration of functional programming paradigm and object-oriented programming paradigm. Java is an integration of imperative programming paradigm, object-oriented programming paradigm, concurrent programming paradigm, event-based programming paradigm and web-based programming.

1.4.2 Implementation-based Classification

Languages can also be classified according to the implementation models, i.e. according to the storage allocation. There are four implementation models:

1. **Static implementation:** In this model, the memory is allocated for every declared data structure and variables at the time of compilation.
2. **Stack-based implementation:** In this model, a control stack is used to allocate the memory for local variables and other run-time requirements, such as various pointers. The advantage of this model is that the stack can grow and shrink at run time according to the memory required by the chain of programs called subprograms.
3. **Heap-based implementation**: This model uses a central common memory area accessible to all procedures. The memory can be borrowed from the heap and released

Languages can also be classified according to the implementation models, i.e. according to the storage allocation. There are four implementation models:

1. **Static implementation:** In this model, the memory is allocated for every declared data structure and variables at the time of compilation.
2. **Stack-based implementation:** In this model, a control stack is used to allocate the memory for local variables and other run-time requirements, such as various pointers. The advantage of this model is that the stack can grow and shrink at run time according to the memory required by the chain of programs called subprograms.
3. **Heap-based implementation**: This model uses a central common memory area accessible to all procedures. The memory can be borrowed from the heap and released dynamically at run time. It allows the allocation of dynamic data structures or sequenced data.
4. **An integrated model that integrates static, stack-based, and heap-based memory**: In this model, all types of memory allocation are integrated for execution efficiency.

1.4.3 Other Classifications

Programming languages can also be classified according to the type of system they support. The two major classifications are — monomorphic and polymorphic. A function in monomorphic language can only handle one type of data structure. However, in polymorphic, only one generic function is required.

Languages, such as FORTRAN, ALGOL Pascal, and C, are monomorphic languages. They support a limited amount of polymorphism. Many modern-day languages, such as C++, Lisp, ML, Haskell, Java and Prolog, are polymorphic languages.

Another way to classify programming language is to identify the domain at which they are best at solving problems. A language that fulfils the requirements of a problem domain can be classified as a language suitable for that domain. For example, C++ is a programming language for large software development and FORTRAN is a good language for scientific programming.

1.5 Role of Programming Languages

Earlier, languages were designed to execute programs efficiently. Computers were expensive and critical resource, and any high-level language was compared with the execution behaviour of assembly language.

In 1960s, after the development of FORTRAN, COBOL, LISP and ALGOL, machines were less expensive and programming costs were higher. There was also a need to move programs from one system to another system. Instead of compiling a program of large and expensive computer, high-level languages were used to develop correct programs to solve problems. Then in 1970s, compilers were matured and languages were focussed on solving domain-specific problems.

1.5.1 Characteristics of Good Language

Some of the characteristics of good language are as follows:

- **Clarity, simplicity and unity:** A programming language provides a conceptual framework to think about algorithms and a means to express those algorithms. The language should help a programmer before the coding stage. It should provide clear, simple and unified concepts that can be used to develop algorithms.

The syntax of a language affects the ease at which a program can be written or modified. It should be easy to read the program. And, a program should be easy to be modified.

- **Orthogonality:** It refers to the attribute of being able to include various features of a language. When the features of a language are orthogonal, it is easy to learn and write programs as there are very few exceptions.
- **Naturalness:** A language needs a syntax that logically reflects the structure of the algorithm. It should be possible to translate such a program design directly to a program statement. The language should provide data structures, operations, control structures and a natural syntax.
- **Support for abstraction:** Every programming language has a substantial gap remaining between the abstract data structures and operations to characterise the solution to a problem.
- **Easy of program verification:** It is necessary that the programs written in a language should be verifiable. There are many ways to verify that if a program is performing

- **Orthogonality:** It refers to the attribute of being able to include various features of a language. When the features of a language are orthogonal, it is easy to learn and write programs as there are very few exceptions.
- **Naturalness:** A language needs a syntax that logically reflects the structure of the algorithm. It should be possible to translate such a program design directly to a program statement. The language should provide data structures, operations, control structures and a natural syntax.
- **Support for abstraction:** Every programming language has a substantial gap remaining between the abstract data structures and operations to characterise the solution to a problem.
- **Easy of program verification:** It is necessary that the programs written in a language should be verifiable. There are many ways to verify that if a program is performing its required function. A program can be corrected by a forma verification method. A language whose program cannot be verified easily will be more troublesome to use.
- **Programming environment:** It is necessary that a programming language has an appropriate programming environment. It makes a reliable, efficient and well-documented language available.
- **Portability of programs:** A language that is widely available and whose definition is independent of the features form a useful base for the production of transportable programs.
- **Cost of use:** The cost of program execution, program translation, program creation, testing and use, and program maintainence should be reasonable.

1.5.2 Introducing Programming Environments

A programming environment is the environment in which programs are created and tested. It tends to have a less influence on language design than the operating environment in which programs are expected to be executed. It consists of a set of support tools and a command language for invoking them. Some of the tools include editors, debuggers, verifiers, test data generators and printers.

1.6 Recap and Questions

Let us quickly recapitulate the following discussion points:

- A program is the specification that is used to process the data and handle data flow between modules.
- A program is a sequence of meaningful symbols to specify the solution of a complex problem.
- A program consists of three components — logic, abstraction and control.
- Programming paradigm is defined as the style of programming.
- Languages can be classified using various criteria, such as programming paradigms, implementation models, the notions of types used in the programming languages and their intended problem domains.
- A programming environment is the environment in which programs are created and tested.

1.6.1 Test Your Skills - Multiple Choice Questions

1. In which of the following programming paradigms, an assertion or assignment statement is used to change the state of the low-level von Neumann machine?

a. Imperative programming paradigm
b. Declarative programming paradigm
c. Object-oriented programming paradigm
d. Concurrent programming paradigm

2. In which of the following programming paradigms, a declarative program consists of logic and abstraction at the programmer level?

a. Imperative programming paradigm
b. Declarative programming paradigm

1.6.1 Test Your Skills - Multiple Choice Questions

1. In which of the following programming paradigms, an assertion or assignment statement is used to change the state of the low-level von Neumann machine?

a. Imperative programming paradigm
b. Declarative programming paradigm
c. Object-oriented programming paradigm
d. Concurrent programming paradigm

2. In which of the following programming paradigms, a declarative program consists of logic and abstraction at the programmer level?

a. Imperative programming paradigm
b. Declarative programming paradigm
c. Object-oriented programming paradigm
d. Concurrent programming paradigm

3. Which of the following is the first language to use object-oriented programming?

a. Algol
b. PASCAL
c. C++
d. SIMULA

4. In which of the following programming languages, visual programming has been used for event-based programming?

a. C#
b. PASCAL

4. In which of the following programming languages, visual programming has been used for event-based programming?

a. C#
b. PASCAL
c. C++
d. SIMULA

5. Which of the following programming paradigms never waits for any input from the external world?

a. Imperative programming paradigm
b. Declarative programming paradigm
c. Object-oriented programming paradigm
d. Event-based programming paradigm

6. What do you understand by side effects of a programming language?

a. Affecting logic
b. Affecting state
c. Affecting facts
d. Affecting rules

7. _____ allocation is used for sequenced data types.

a. Stack
b. Static
c. Heap
d. None

d. None

1.6.2 Fill in the Blanks

1. A _____ is a sequence of meaningful symbols to specify the solution of a complex problem.
2. A program consists of three components – logic, _____ and control.
3. _____ is the high-level specification of the solution to a problem.
4. _____ is defined as modelling an entity with the desired attributes required to solve the problem.
5. _____ is defined as mapping the solution of the problem.

1.6.3 Theoretical Questions

1. What do you mean by a program?
2. Define abstraction in programs.
3. Explain the major criteria of program execution.
4. Explain the characteristics of a good language.
5. What do you mean by a programming environment?
6. Explain various programming paradigms.

2.1 Learning Names, Scopes and Bindings

Learning Objectives

At the end of this module, you will be able to:

- Explain scope rules and how to implement scope.
- Describe meaning of names within a scope.
- Define binding of referencing environment.
- Define macro expansion.
- Explain object lifetime and storage management

2.1 Introduction

Higher-level programming languages are known as high-level because they provide a higher degree of abstraction. The abstraction in this case refers to the degree to which language features are separated from the details of any particular computer architecture. Languages, such as Fortran, Algol, and Lisp, were developed to achieve machine independence and ease of programming. With the help of abstraction, designers were able to write programs that run on various machines and are easy to understand.

A name is a mnemonic character string that is used to do something else. Though it is an identifier, symbols, such as + or :=, can also be names. Names are used to refer variables, constants, operations, types, etc. These elements in a programming environment would have required memory addressing if names were not used. Thus, names also

- Explain scope rules and how to implement scope.
- Describe meaning of names within a scope.
- Define binding of referencing environment.
- Define macro expansion.
- Explain object lifetime and storage management

2.1 Introduction

Higher-level programming languages are known as high-level because they provide a higher degree of abstraction. The abstraction in this case refers to the degree to which language features are separated from the details of any particular computer architecture. Languages, such as Fortran, Algol, and Lisp, were developed to achieve machine independence and ease of programming. With the help of abstraction, designers were able to write programs that run on various machines and are easy to understand.

A name is a mnemonic character string that is used to do something else. Though it is an identifier, symbols, such as + or :=, can also be names. Names are used to refer variables, constants, operations, types, etc. These elements in a programming environment would have required memory addressing if names were not used. Thus, names also provide abstraction. Subroutines are control abstractions. They allow the programmer to hide the complicated code behind a simple interface.

In this chapter, you will learn the scope rules that define the region where most name-to-object bindings are usable. Then, you will learn the scope rules where a referencing environment may be bound to a subroutine that is passed as a parameter, returned from a function, or stored in a variable. You will learn about aliasing, overloading and macro expansion. At the end of this chapter, you will also learn various mechanisms used to allocate and deallocate storage space for object.

2.2 Learning Scope Rules and Implementing Scope

In general, a binding can be defined as an association between two things, such as a name and the thing it names. The binding time is the time when it is created or at the time when any implementation decision is made. The region in the program where a binding is activated is scope. In modern programming languages, the scope of a binding is determined statically during the compile time. In C, a new scope is created during the start of a subroutine. Then, the binding for local objects are created and bindings for global objects are deactivated. After exiting the subroutine, the bindings for local variables are destroyed and bindings for global objects are reactivated. In a C program, it is possible to determine which names refer to which objects at which points. Therefore, C is in the program statically scoped. Languages, such as APL, Snobol and early versions of Lisp, are dynamically scoped. It means that their bindings depend on the flow of execution at run time.

Basically, a scope is a program region of maximal size where no bindings change. It is the body of a module, class, subroutine, or structured control flow statement. It is sometimes known as a block. In C language, it is defined by {...} braces. The set of active bindings during a given point in a program's execution is known as the current referencing environment. It is determined by static or dynamic scope rules. In some case, the referencing environment depends on the binding rules. When a reference to a subroutine S is stored as a variable and then passed as a parameter to another subroutine, it is required to find when the referencing environment for S is selected.

Example 1: C++ Code to explain scope of the variable

```
1.    int i = 25;            //This is a global variable

2.    int call1();

3.    int main(){

4.          cout<<i<<endl;
```

```
        int i = 50;     //This is the local variable
        cout<<i<<endl;
        call1();
    }
    int call1(){
        cout<<i;
    }
```

Output:

```
25
50
25
```

Output:

```
25

50

25
```

In this example, the variable *i* declared in line number 1 is the global variable. It is available throughout the program unless a local variable overwrites it, i.e. its scope is the entire program. The variable *i* declared in line number 5 is the local variable and is available only in the main function, i.e. its scope is limited to the function in which it is defined. Note that the scope begins from the declaration and ends with the end of the function, i.e. from line number 5 to 8 except for any subroutine calls. Hence, the output of the code at line number 4 results in *25* whereas the output of the code at line number 6 is *50*. When the call function is called at line number 7, the global variable again comes into effect in the lines 9, 10 and 11. Hence, the output of line number 10 is again *25*.

2.2.1 Scope Rules

The scope rules define and guide the visibility of entities in the program. The three rules are listed below:

1. The scope of an entity is the program or function in which it is declared.

```
int main()
        {
                int i=4;
```

2.2.1 Scope Rules

The scope rules define and guide the visibility of entities in the program. The three rules are listed below:

1. The scope of an entity is the program or function in which it is declared.

```
int main()
        {
                int i=4;
                cout<<i;
        }
```

   In this C++ snippet, the scope of variable *i* is the *main* function.

2. A global entity is visible to all contained functions, including the function in which that entity is declared.

```
        int i =1;
        int main()
        {
                cout<<i;
```

```
        cout<<i;
}
```

In this C++ snippet, *i* is a global variable and, hence, is available for use in all the functions listed in the program file including main function.

3. An entity declared in the scope of another entity is always a different entity even if their names are identical.

```
int main()
{
        int i=4;
        for(int i=1;i<10;i++);
        cout<< i;
}
```

In this C++ snippet, the output of the cout statement will be 4 printed once. This is because the for loop declares its own copy of variable *i*, whose scope remains in the for loop and, hence, after the for loop the next statement will refer to the variable *i* declared before the loop.

[Note: The for loop in the snippet performs No Operation because of the semi colon attached at the end of the declaration.]

[Note: The for loop in the snippet performs No Operation because of the semi colon attached at the end of the declaration.]

2.2.3 Static Scoping

In a language that has static scoping, the binding between names and objects can be found at compile time by examining the text of the program. Now, the “current” binding for a given name can be found at the matching declaration whose block most closely surrounds a given point in the program.

The scope rules were very simple in Basic. They became somewhat complex in Fortran, which introduced the differences between global and local variables. The scope of a local variable is limited to the subroutine and the scope of global variables in Fortran can be partitioned into common blocks, which are “imported” by subroutines.

The lifetime of a local Fortran variable is only the single execution of the variable’s subroutine. This rule can be overridden by an explicit save statement. In C, it is possible by declaring the variable static. This variable has a lifetime of the entire execution of the program. Example 1 shows the code that uses the static variables. The subroutine sub_name can be used to series of character-string names — A1, A2, A3…

Example 1: C Code to use Static Variables

```
void sub_name (char *s) {
    static short int i; /*This guarantees that static locals are
    initialised to zero*/
```

Example 1: C Code to use Static Variables

```
void sub_name (char *s) {
    static short int i; /*This guarantees that static locals are
    initialised to zero*/
    sprintf (s, "A%d\0", ++i); /* It prints the formatted output
    to s */
}
```

2.2.3 Dynamic Scoping

In a language that has dynamic scoping, the bindings between names and objects depend on the flow of control at run time. They depend on the order in which subroutines are called. In dynamic scoping, the “current” binding for a given name is the most recently occurred during the execution and is not yet destroyed by returning from its scope.

Languages that have dynamic scoping include APL, Snobol, TEX, early dialects of Lisp and Perl. Since the flow of control cannot be predicted, the bindings between names and

2.2.3 Dynamic Scoping

In a language that has dynamic scoping, the bindings between names and objects depend on the flow of control at run time. They depend on the order in which subroutines are called. In dynamic scoping, the “current” binding for a given name is the most recently occurred during the execution and is not yet destroyed by returning from its scope.

Languages that have dynamic scoping include APL, Snobol, TEX, early dialects of Lisp and Perl. Since the flow of control cannot be predicted, the bindings between names and objects in language with dynamic scoping cannot be determined by a compiler. Therefore, the languages with dynamic scoping are interpreted rather than compiled.

In Example 2, if static scoping is used then it prints 1 and if dynamic scoping is used, the output then depends on the value at line 8 at run time. If the input is positive, the program prints 2; otherwise it prints 1. The issue is if the assignment to the variable at line 3 refers to global variable at line 1 or local variable at line 5. According to static scope rules, the reference is with the closest declaration that is global n. However, according to dynamic scope rules, the most recent and active binding for n is used, which is local variable. The method at line 12 and the assignment at line 7 both always refer to global variable. For static scope, when the first method is called, it also affects the global variable and hence the output of line 12 is 2. For dynamic scoping when positive input is given, the assignment inside the function first will refer to the local variable declared inside function second. Hence, the local variable would take the value as 2 and the global variable remains unaffected as 1. As a result, the output is 1.

Example 2: Static versus Dynamic Scoping

```
1. n : integer – global declaration

2. procedure first

3.   n:=1.

4. procedure second
```

```
procedure second
  n : integer – local declaration
  first()
n:=2
if read_integer()>0
  second()
else
  first()
write_integer(n)
```

In dynamic scoping, it is possible that errors may occur with the referencing environment. They may not be detected until run time. The best thing about dynamic scoping is that it facilitates the customisation of subroutines.

Thus, static scoping, i.e. early binding, is said to provide greater efficiency whereas dynamic binding, i.e. late binding, is said to provide greater flexibility. Following table

Thus, static scoping, i.e. early binding, is said to provide greater efficiency whereas dynamic binding, i.e. late binding, is said to provide greater flexibility. Following table summarizes the difference between static and dynamic binding:

Table 1: Static vs Dynamic Binding

Static Scoping	**Dynamic Scoping**
Variables and function call information is available at compile time	Variables and function call information is determined at run time
Greater efficiency	Lower efficiency
Less flexibility	Greater flexibility
Variable scope is already determined	Most recent variable scope is considered

2.3 Modules

A challenge while creating a large software is how to divide the effort among programmers. This depends on information hiding that makes objects and algorithms invisible. A well-designed program has a “narrow” interface between the modules and any change decision is simply hidden inside a single module. Properly modularised code reduces the load on the programmer by minimising the amount of information required to understand any given module. Information hiding also helps in reducing the name conflicts as well as safeguards the integrity of data abstractions. It also helps to compartmentalise run-time errors.

Static variable allows a subroutine to have memory to retain the information from one invocation to the next. In another meaning, static variables allow the programmers to build single-subroutine abstractions. However, it is not possible to create the abstractions for the interface that has more than one subroutine. It can be possible in many languages using a module construct.

2.3.1 Modules as Abstractions

A module consists of a collection of objects, such as subroutines, variables, types, etc., to be encapsulated in such a way that:

- the objects inside will be invisible to each other
- the objects inside will not be visible to outside unless they are explicitly exported
- the objects outside will not be visible to inside unless they are explicitly imported

Modules appear in Haskell, C++, Java, and C# and in major scripting languages. Several languages, such as Ada, Java, and Perl, use the term package instead of module. Other languages, such as C++, C# and PHP, use namespace.

2.3.2 Imports and Exports

2.3.2 Imports and Exports

Most of the module-based languages allow a programmer to specify certain exported names in a restricted manner. Variables can be exported as only read-only. Modules where names must be explicitly imported are known as closed scopes and modules that do not need imports are known as open scopes.

Modules are closed in Modula (1, 2, and 3) and Haskell, and in Ada, Java, C#, and Python, they have selectively open scopes. In these languages, a name foo exported from module A is automatically visible in module B as A.foo.Module Types and Classes

In Simula, Euclid and ML, modules are treated as types instead of simple encapsulation constructs. According to the module type, a programmer can declare an arbitrary number of similar module objects. Many languages also provide a class construct for object-oriented programming. Classes are considered as module types that are augmented with an inheritance mechanism. Inheritance allows new classes to be defined as an extension of existing classes. It provides a programming style that allows new objects to inherit operations from existing objects without rewriting the code. Classes are commonly used in object-oriented languages, such as Smalltalk, Eiffel, C++, Java and C#. They are also used in some scripting languages, such as Python and Ruby.

2.3.3 Nested Subroutines

Some languages, such as Pascal, Ada, ML, Python, Scheme, Common Lisp and Fortran 90, use nested subroutines. Other languages, such as C, allow classes or scopes to next. Most languages provide a collection of predefined objects and types, such as integer and char. These objects are declared in an invisible outermost scope that surrounds the scope in which global objects are declared. The outermost scope allows a programmer to define a global object whose name is same to the name of some predefined object. A name-to-object binding which is hidden by a nested declaration of same name have a hole in its scope. In languages where the object name is hidden is inaccessible in the nested scope. In C++ where subroutines are not allowed to nest, ::A is used to refer the global declaration of A, regardless if the current subroutine also has an A.

2.3.4 Declaration Order

2.3.4 Declaration Order

Several early languages, such as Algol 60 and Lisp, need that the declarations should be made at the beginning of their scope. Languages, such as C++ and Java, do not need to define the declarations before use. In these languages, members of a class are visible inside the methods of the class. In Java, classes can be declared in any order. Recursive type and subroutines can create a problem for the languages where the names should be declared before they can be used. In C and C++, this problem is handled by differentiating the declaration of an object and its definition. A declaration includes a name and its scope but not implementation, but a definition includes the object with details needed for its implementation. Example 3 shows the declarations and definitions in C:

Example 3: Declarations and Definitions in C

```
struct teacher;              /* declaration only */

struct student {

      struct teacher *principal;

      struct student *next_student;

      ...

};
```

```
struct teacher {            /* definition */
      struct student *first_student;
      ...
};
```

The initial declarations of teacher are required to introduce a name. Since the pointers are of same size, the compiler can find the implementation of student without knowing any teacher details. In some languages, a nester declaration is hidden in the outer declaration of the same name. Example 4 shows a C program where variables are declared in nested blocks:

Example 4: Declarations in Nested Block

```
{
      int temp = a;
      a = b;
      b = temp;
```

```
        b = temp;

    }
```

2.3.5 Meaning of Names within a Scope

Till now, you have learnt that there is only one-to-one mapping between names and visible objects in any given point in a program. This might not be always the case. Two or more names can refer to the same object at the same point in a program are known as aliases. A name that is referring to more than one object at a point in a program is known as overloaded.

2.3.6 Aliases

Some examples of aliases can be found in the common blocks and equivalence statements of Fortran. They are found in the programs that use pointer-based data structures. A way to create aliases is by passing a variable by reference to a subroutine that can access that variable directly. Example 5 shows a C++ code to create aliases with parameters:

Example 5: Create Aliases with Parameters

```
double add, add_of_cubes;

...
```

```
void accumulate(double& x)      // x is passed by reference
{
      add += n;
      add_of_cubes += n*n*n;
}
...
accumulate(add);
```

If add is passed as an accumulate, then add and n will be aliases for one another. This will cause the program to not do what it is supposed to.

2.3.7 Overloading

Most programming languages support a limited form of overloading. For example, in C, the plus sign (+) is used to name various functions that include signed and unsigned integer and floating-point addition. In C, C++ and Pascal, it is not possible to overload enumeration constants. Every constant that is visible in the given scope should be distinct. Ada and C++ provide ways to overload the subroutine names. Example 6 shows the overloading in C++:

2.3.7 Overloading

Most programming languages support a limited form of overloading. For example, in C, the plus sign (+) is used to name various functions that include signed and unsigned integer and floating-point addition. In C, C++ and Pascal, it is not possible to overload enumeration constants. Every constant that is visible in the given scope should be distinct. Ada and C++ provide ways to overload the subroutine names. Example 6 shows the overloading in C++:

Example 6: Overloading in C++

```
struct complex {
      double real, imaginary;
};
enum base {dec, bin, oct, hex};


int n;
complex x;
```

```
accumulate(add);
```

If add is passed as an accumulate, then add and n will be aliases for one another. This will cause the program to not do what it is supposed to.

2.3.7 Overloading

Most programming languages support a limited form of overloading. For example, in C, the plus sign (+) is used to name various functions that include signed and unsigned integer and floating-point addition. In C, C++ and Pascal, it is not possible to overload enumeration constants. Every constant that is visible in the given scope should be distinct. Ada and C++ provide ways to overload the subroutine names. Example 6 shows the overloading in C++:

Example 6: Overloading in C++

```
struct complex {
      double real, imaginary;
};
enum base {dec, bin, oct, hex};

int n;
complex x;

void print_num(int i) {...
void print_num(int i, base b) { ...
void print_num(complex c) { ...
print_num(n);        // uses the first function above
print_num(n, hex);   // uses the second function above
print_num(x);        // uses the third function above
```

In Ada, C++, C#, Fortran 90 and Haskell, the built-in arithmetic operators (+,-,*, etc.) can also be overloaded with user-defined functions. In C++ and C#, A+B can be short for either operator+(A, B) or A.operator+(B).

2.4 Binding of Referencing Environments

Earlier, you have learnt how scope rules are used to determine the referencing environment of a given statement in a program. Static scope rules specify that the referencing environment should depend on the textual nesting of program blocks where names are declared. Dynamic scope rules specify that the referencing environment should depend on the order where declarations are encountered at run time.

The late binding of the referencing environment of a subroutine that has been passed as a parameter is known as shallow binding. It is found in languages that have dynamic scoping. In situations, where the code that originally passes the function as a parameter has a particular referencing environment, it also does not want the routine to be called in any other environment. It binds the environment when the routine is first passed as a parameter and restores the environment when the routine is finally called. This early binding of the referencing environment is known as deep binding.

2.4.1 Subroutine Closures

Deep binding is implemented by creating a referencing environment and combining it together with a reference to the subroutine. This bundle is referred to as a closure. Normally, the subroutine can be represented in the closure by a pointer. In languages that use dynamic scoping, the referencing environment depends on if the language implementation uses an association list or a central reference table for run-time lookup of names.

Although shallow binding is usually used as the default in languages with dynamic scoping, deep binding can be used as well. In early Lisp, the built-in primitive function takes a function as its argument and returns a closure whose referencing environment would be used if the function is called at the present time.

Deep binding is the default in the languages that have static scoping. A closure in a language that has static scoping collects the instance of every object when the closure is created. When the closure's subroutine is called, it finds the captured instanced even if newer instances have subsequently been created by recursive calls.

You should remember that binding rules matter with static scoping only while accessing objects that are neither local nor global. If an object is local to the current subroutine, then it does not matter if the subroutine was called directly or through a closure. If an object is global, there will be only a single instance because the main body is not recursive. Therefore

You should remember that binding rules matter with static scoping only while accessing objects that are neither local nor global. If an object is local to the current subroutine, then it does not matter if the subroutine was called directly or through a closure. If an object is global, there will be only a single instance because the main body is not recursive. Therefore, binding rules in languages, such as C, are irrelevant as it has no nested subroutines.

2.4.2 Object Closures

The referencing environment in a closure will be important only while passing a nested subroutine. It means that the implementation of first-class routines is unimportant in a language that does not have nested subroutines. In object-oriented languages, another way to pass a subroutine with a context is by encapsulating the subroutine as a method of a simple object. Example 7 shows an example of an object closure in Java:

Example 7: Object Closure in Java

```
interface IntFunc {
      public int call(int a);
}
class PlusB implements IntFunc {
      final int b;
```

```
        PlusB(int c) { b=c; }

        public int call(int a) { return a+b;}

    }

    ...

    IntFunc f1 = new PlusB(2);

    System.out.println(f1.call(3));      // prints 5
```

In Example 7, the interface IntFunc defines a static type for objects enclosing a function from integers to integers. Class PlusB is an implementation of this type and can be instantiated for a constant b. The Java code in this example captures b in the object closure returned by new PlusB(2).

An object that can be used as a function and its referencing environment can be known as an object closure, a function object, or a functor. Object closures are important in languages that support special syntax. Example 8 shows an object of a class that overrides operator() and can be called as if it were a function:

Example 8: Function Objects in C++

```
class int_func {

public:
```

```
    virtual int operator()(int a) = 0;
};
class plus_b : public int_func {
    const int b;
public:
    plus_b(int c) : b(c) { }
    virtual int operator()(int a) { return a+b;}
};
...
plus_b f(2);
cout << f(3) << "\n";       // prints 5
```

2.5 Macro Expansion, Object Lifetime and Storage Management

Earlier when high-level programming languages were not developed, assembly language programmers had to write a highly repetitive code. So, many assemblers then provided macro expansion facilities. Let us take a task of loading an element of a two-dimensional array from memory into a register. This operation may need a lot of instructions depending on hardware instruction set, the size of the array elements and where indices are constants. In assemblers, a macro can be used to replace an expression with the appropriate multi-instruction sequence.

When C was created, pre-processor macros, as shown in Example 9, were included:

Example 9: Preprocessor Macro in C

```
#define LINE_LEN 100

#define DIVIDES(x,n)  (!((n) % (x)))

     /* true if n has zero remainder modulo a */

#define SWAP(x,y) {int temp = (x); (x) = (y); (y) = temp;}

#define MAX(x,y) ((x) > (y) ? (x) : (y))
```

Macros, such as LINE_LEN, are used to support named constants. Sometimes, the parameterised macros, such as DIVIDES, MAX, and SWAP, are more efficient than the equivalent C

Macros, such as LINE_LEN, are used to support named constants. Sometimes, the parameterised macros, such as DIVIDES, MAX, and SWAP, are more efficient than the equivalent C functions. It helps in avoiding the overhead of the subroutine call mechanism.

However, C macros have many limitations because they are implemented using as a textual substitution and are not understood by the rest of the compiler. For example, in case of the DIVIDES, the parenthesis around x and y are important. Without them DIVIDES(b + c, a) would be similar to (!(a %b + c)) and (!((a % b) + c)).

2.5.1 Object Lifetime and Storage Management

While discussing names and bindings, it is necessary to specify a difference between the names and objects, and to identify the following key events:

- Creation of objects
- Creation of bindings
- References to variables, subroutines, types, etc. that use bindings
- Deactivation and reactivation of bindings
- Destruction of bindings
- Destruction of objects

When a name-to-object binding is created and destroyed, it is known as the binding's lifetime. In the same manner, when an object is created and destroyed, it is known as the object's lifetime. When an object retains the value, then it can be accessed even when name cannot used to access it. When a variable is passed to a subroutine by reference, then the lifetime binding between the parameter and variable is shorter than the variable itself. A binding to an object that is no longer live is known as dangling reference.

The lifetime of an object is usually corresponding to one of the following three principal storage allocation methods that are used to manage the space of an object:

- **Static Allocation**: It means that objects have an absolute address which is retained throughout the program's execution.
- **Stack-based Allocation**: It means that objects are allocated and deallocated in last-in, first-out order along with the subroutine calls and returns.
- **Heap-based Allocation**: It means the objects can be allocated and deallocated at arbitrary times and they need a storage management algorithm.

- **Static Allocation**: It means that objects have an absolute address which is retained throughout the program's execution.
- **Stack-based Allocation**: It means that objects are allocated and deallocated in last-in, first-out order along with the subroutine calls and returns.
- **Heap-based Allocation**: It means the objects can be allocated and deallocated at arbitrary times and they need a storage management algorithm.

2.5.2 Static Allocation

Global variables are an example of static objects. Other examples of statically allocated objects are numeric and string-valued constant literals, such as A = B/14.7 or printf("hello, world\n"). Some compilers also use various tables that are used by run-time support routines for debugging, dynamic-type checking, garbage collection, exception handling, and other purposes. These tables are also statically allocated.

Statically allocated objects are those whose value should not change during the execution of program. These objects are also allocated in protected, read-only memory, so that it is not possible to write them and it doesn't generate a run-time error.

Normally, local variables are created when their subroutine is called and destroyed when it returns. If it is being called repetitively, then each invocation creates and destroys a separate instance of each local variable.

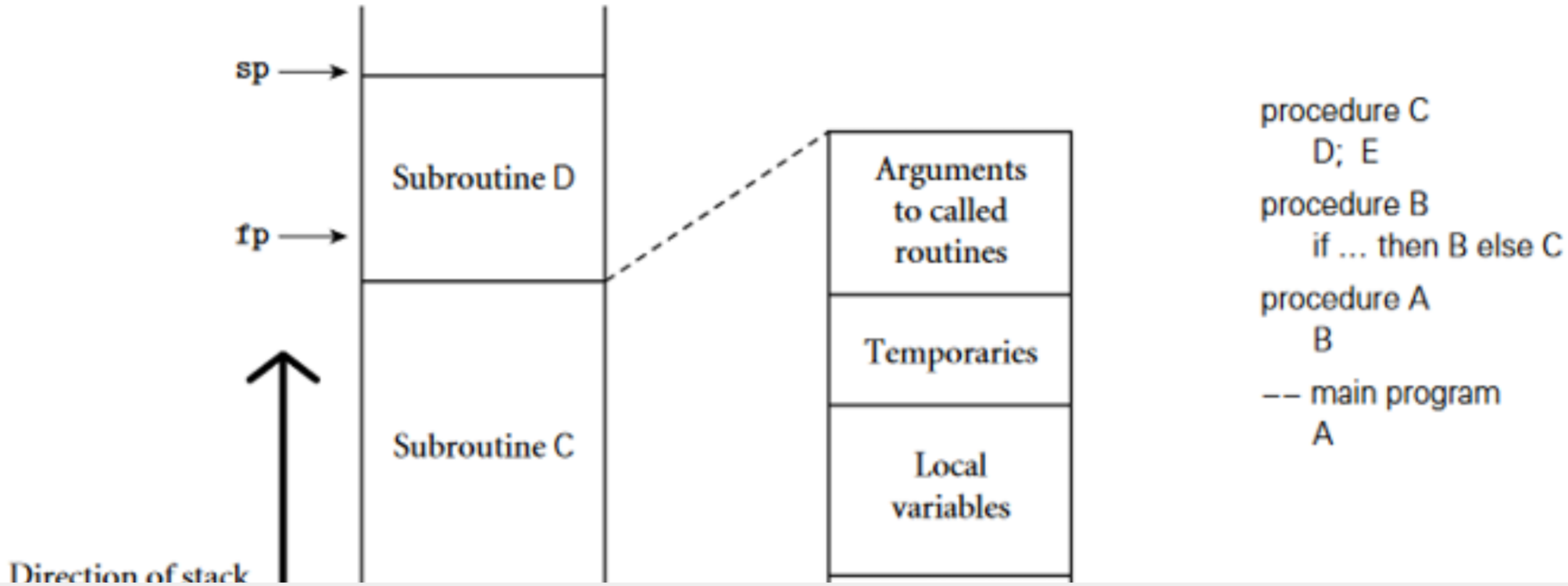
In various languages, a constant is required to have a value that is to be determined at compile time. Normally, the expression that is used to specify the value of constant is allowed to only include other known constants and built-in functions and arithmetic operators. These name constants are known as manifest or compile-time constants. They are allocated statically even if they are local to a subroutine.

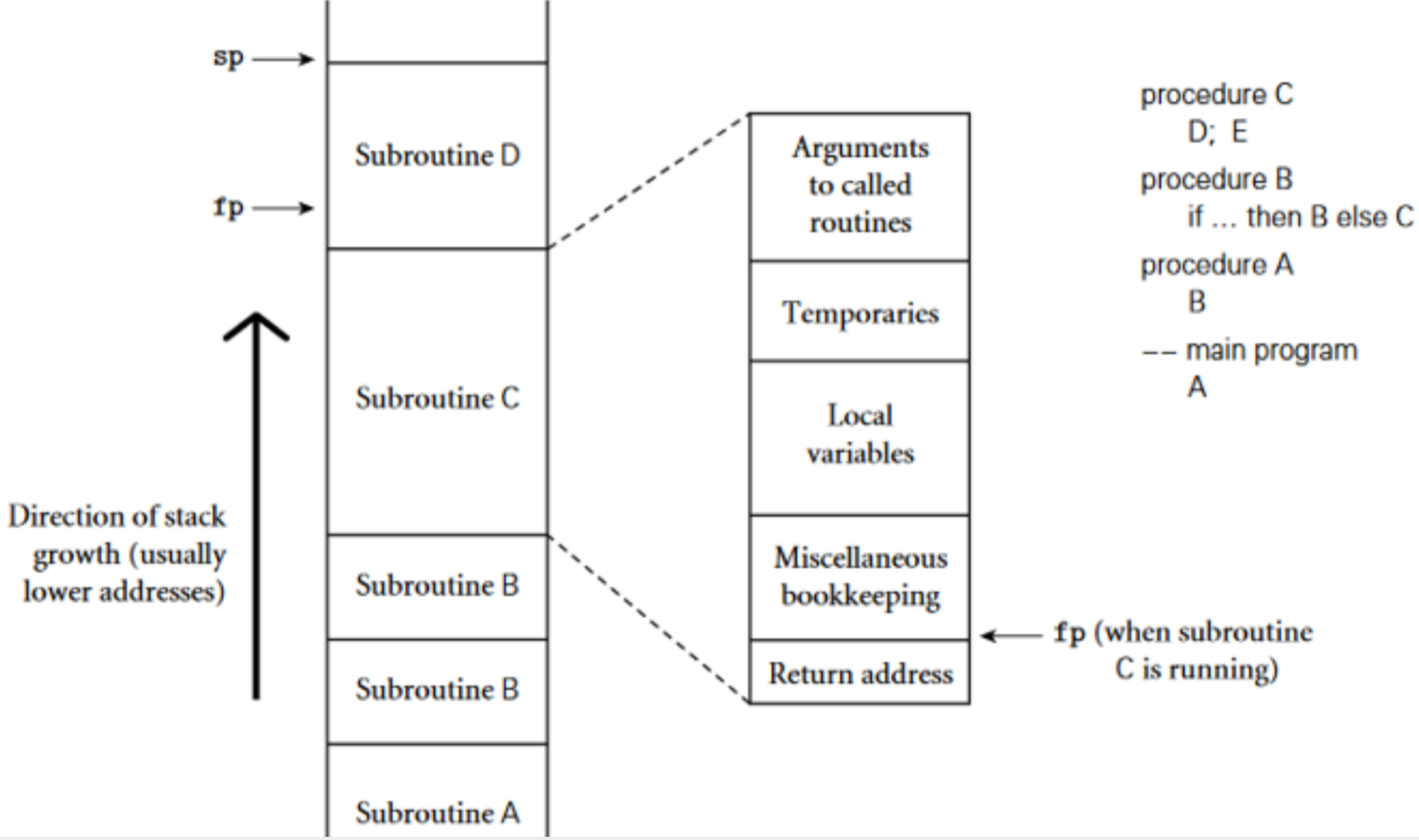
In some languages, such as C and Ada, constants are just variables that cannot be changed. Their values sometimes depend on other values that are not known until run time. These constants when local to a recursive subroutine must be allocated on the stack. Along with the local variables, the compiler stores a variety of other information related to the subroutine. It includes arguments and return values, temporaries and bookkeeping information.

2.5.3 Stack-based Allocation

2.5.3 Stack-based Allocation

If a language allows recursion, then the static allocation of variable cannot be done because the number of instance of a variable that are required at a time can be conceptually unbounded. Therefore, nesting of subroutine calls is used to allocate space for locals on a stack. Figure 1 shows a picture of a typical stack:




sp
fp
Subroutine D
Subroutine C
Subroutine B
Subroutine B
Subroutine A
Direction of stack growth (usually lower addresses)
Arguments to called routines
Temporaries
Local variables
Miscellaneous bookkeeping
Return address
fp (when subroutine C is running)
procedure C
D; E
procedure B
if ... then B else C
procedure A
B
-- main program
A

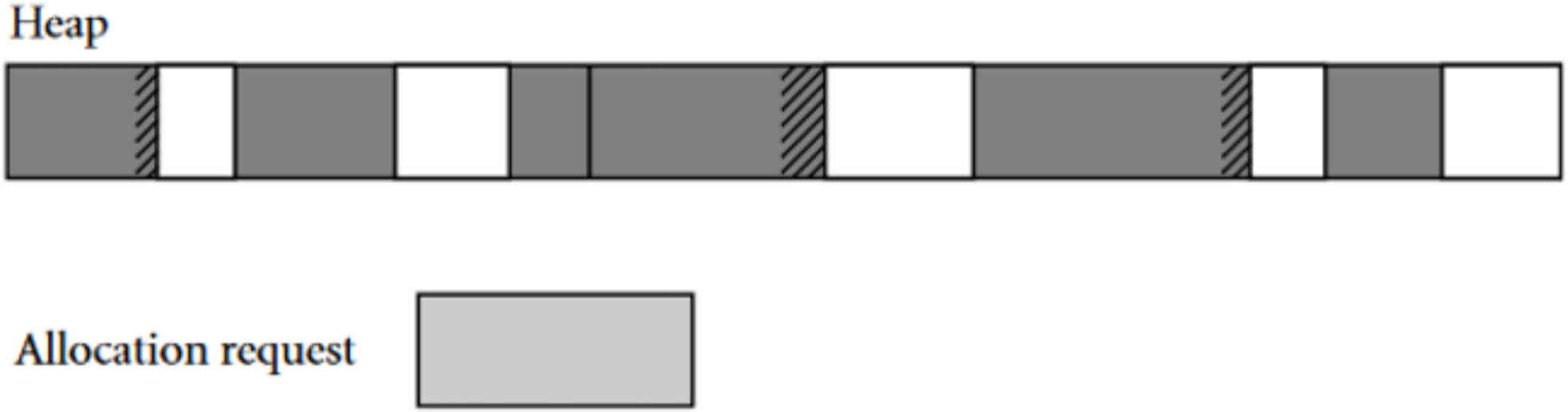
Example 10: Stack-based Allocation

Each instance on this subroutine at runtime has its own frame on the stack that contains arguments and return values, local variables, temporaries, and bookkeeping information. Arguments that are passed to the subsequent routines are located at the top of the frame.

The subroutine calling sequence — the prologue and epilogue — is responsible for the maintenance of the stack. The calling sequence is the code executed by the caller immediately before and after the call. The prologue refers to the code executed at the beginning and the epilogue is the code executed at the end. While it is not possible to predict the location of a stack, the offsets of objects withing a frame can be statically determined. The compiler can find a register as frame pointer to always point to a known location within the frame of the current subroutine.

2.5.4 Heap-based Allocation

A heap can be defined as a region of a storage where the subblocks can be allocated and deallocated at arbitrary times. They are required for dynamically allocated pieces of linked data structures and objects, such as character strings, lists and sets. Internal fragmentation happens when a storage-management algorithm allocates a block that is larger than required to hold a given object. In the figure below, the hatched lines show internal fragmentation. External fragmentation happens when the blocks that have been assigned to active objects are scattered through the heap in such a way that the remaining unused space is composed of multiple blocks, such as the white blocks in the figure and the arrived allocation request. Figure 2 shows the fragmentation:



Example 11: Fragmentation

Most storage-based management algorithms use a single linked list of heap blocks that are not currently in use. The list consists of a single block of the entire heap. At each allocation, the algorithm search for a block of an appropriate size. The first fit algorithm selects the first block on the list that has proper size. The best fit algorithm selects the entire list that is smallest enough to satisfy the request.

Normally, it is possible that the best fit algorithm will be best for reserving the blocks for large requests. However, it has a higher allocation cost than a first fit algorithm as it always searches the entire list and has a large number of small remaining blocks.

2.5.5 Different Binding Times

2.5.5 Different Binding Times

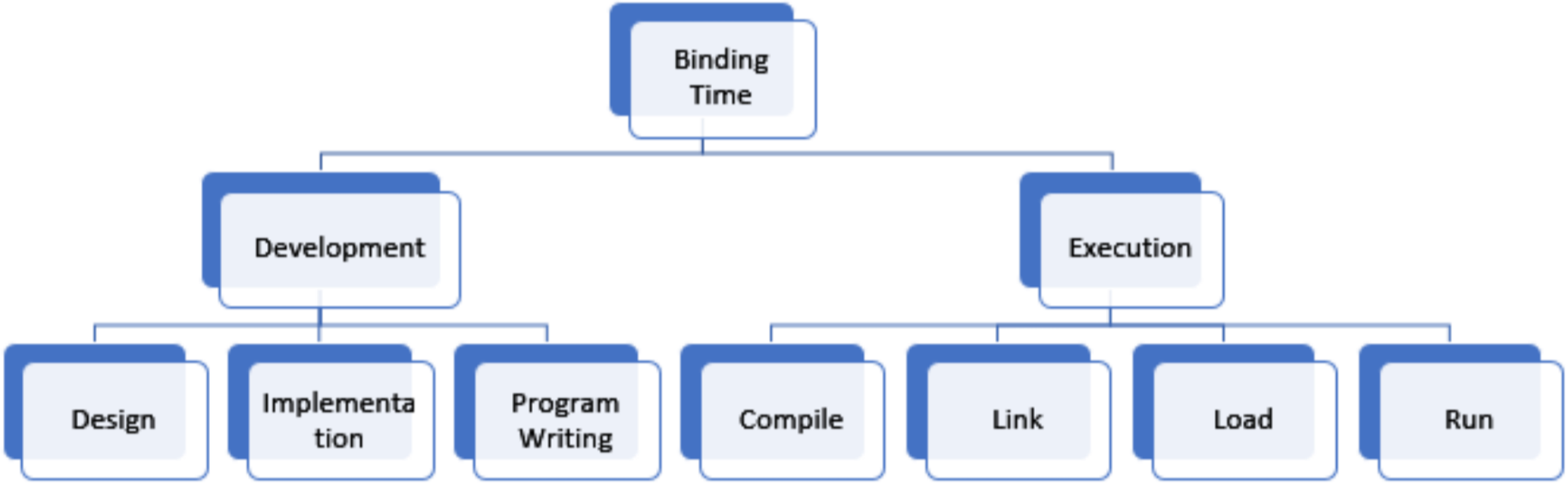


Figure 1: Different Binding Times

Figure 1: Different Binding Times

1. Language design time: In most languages, the control flow constructs, the set of fundamental (primitive) types, the available constructors for creating complex types and many other aspects of language semantics are chosen when the language is designed.

2. Language implementation time: Most language manuals leave a variety of issues to the discretion of the language implementor. Typical examples include the precision of the fundamental types, the coupling of I/O to the operating system's notion of files, the organisation and maximum sizes of stack and heap, and the handling of run-time exceptions, such as arithmetic overflow.

3. Program writing time: Programmers, of course, choose algorithms, data structures and names.

4. Compile time: Compilers choose the mapping of high-level constructs to machine code, including the layout of statically defined data in memory.

5. Link time: Since most compilers support separate compilation — compiling different modules of a program at different times — and depend on the availability of a library of standard subroutines, a program is usually not complete until the various modules are joined together by a linker. The linker chooses the overall layout of the modules with respect to one another and resolves inter-module references. When a name in one module refers to an object in another module, the binding between the two is not finalised until link time.

6. Load time: This refers to the point at which the operating system loads the program into memory so that it can run. In primitive operating systems, the choice of machine addresses for objects within the program was not finalised until load time. Most modern operating systems distinguish between virtual and physical addresses. Virtual addresses are chosen at link time while physical addresses can change at run time. The processor's memory management hardware translates virtual addresses into physical addresses during each individual instruction at run time.

7. Run time: This is actually a very broad term that covers the entire span from the beginning to the end of execution. Bindings of values to variables occur at run time, as do a host of other decisions that vary from language to language.

2.6 Recap and Questions

2.6.1 Recap

Let us quickly recapitulate the following discussion points:

- A binding in general can be defined as an association between two things, such as a name and the thing it names.
- Languages, such as APL, Snobol and early versions of Lisp, are dynamically scoped.
- A scope is a program region of maximal size where no bindings change.
- In a language that has static scoping, the binding between names and objects can be found at compile time by examining the text of the program.
- In a language that has dynamic scoping, the bindings between names and objects depend on the flow of control at run time.
- Information hiding helps in reducing the name conflicts and also safeguards the integrity of data abstractions.
- A name that refers to more than one object at a point in a program is known as overloaded.
- Two or more names can refer to the same object at the same point in a program are known as aliases.
- Deep binding is implemented by creating a referencing environment and combining it together with a reference to the subroutine.
- In assemblers, a macro can be used to replace an expression with the appropriate multi-instruction sequence.
- Static allocation means that objects have an absolute address which is retained throughout the program's execution.

2.6.2 Test Your Skills-Multiple Choice Questions

1. Which of the following languages do not have dynamic scoping?

2.6.2 Test Your Skills-Multiple Choice Questions

1. Which of the following languages do not have dynamic scoping?

a. APL
b. Snobol
c. TEX
d. C

2. Which of the following languages uses the term package instead of module?

a. Haskell
b. C++
c. C#
d. Ada

3. Which of the following has two or more names that can refer to the same object at the same point in a program?

a. Alias
b. Overloaded
c. Static
d. Dynamic

4. Which of the following is known as the late binding of the referencing environment of a subroutine that has been passed as a parameter?

a. Static binding
b. Dynamic binding

4. Which of the following is known as the late binding of the referencing environment of a subroutine that has been passed as a parameter?

a. Static binding
b. Dynamic binding
c. Shallow binding
d. Deep binding

5. In which of the following, objects have an absolute address which is retained throughout the program's execution?

a. Static allocation
b. Stack-based allocation
c. Heap-based allocation
d. Dynamic allocation

2.6.3 Fill in the Blanks

1. The set of active bindings during a given point in a program's execution is known as the _____.
2. Modules where names must be explicitly imported are known as _____ scopes and modules that do not need imports are known as _____ scopes.
3. A name that refers to more than one object at a point in a program is known as _____.
4. An object that can be used as a function and its referencing environment can be known as an _____.
5. _____ binding is implemented by creating a referencing environment and combining it together with a reference to the subroutine.

2.6.4 Theoretical Questions

1. Explain the difference between a declaration and a definition. Why is the distinction important?

c. Heap-based allocation
d. Dynamic allocation

2.6.3 Fill in the Blanks

1. The set of active bindings during a given point in a program's execution is known as the _____.
2. Modules where names must be explicitly imported are known as _____ scopes and modules that do not need imports are known as _____ scopes.
3. A name that refers to more than one object at a point in a program is known as _____.
4. An object that can be used as a function and its referencing environment can be known as an _____.
5. _____ binding is implemented by creating a referencing environment and combining it together with a reference to the subroutine.

2.6.4 Theoretical Questions

1. Explain the difference between a declaration and a definition. Why is the distinction important?
2. Describe the difference between static and dynamic scoping.
3. What is operator overloading?
4. Describe the difference between deep and shallow binding of referencing environments.
5. List the objects and information commonly found in a stack frame.

Chapter 3: Type Systems and Checking

Learning Objectives

At the end of this module, you will be able to:

- Explain type systems and type checking
- Discuss records and variants
- Explain arrays and strings
- Discuss pointers
- Explain equality testing and assignment

3.1 Introduction

Data is of immense importance in any field; be it computers, engineering or business. Data is information in raw form. Data is processed and worked upon to get information. Data may be of different types. It may be text, numeric or in decimal values. The different types of data are known as data types. Data types are an important factor in any programming language.

There is a need for different types of variables to store information, while writing a code in any programming language. Variables do nothing but just reserve memory to store values at some location in the computer. However, you are not required to know the exact location of the variable. You just need to know the two things about a variable that are name of the variable and the type of data it can store.

In this chapter, you will learn about the meaning and purpose of types. You will learn about type equivalence and type compatibility. You will also learn about composite types such as records, arrays, string and pointers.

Chapter 3: Type Systems and Checking

Learning Objectives

At the end of this module, you will be able to:

- Explain type systems and type checking
- Discuss records and variants
- Explain arrays and strings
- Discuss pointers
- Explain equality testing and assignment

3.1 Introduction

Data is of immense importance in any field; be it computers, engineering or business. Data is information in raw form. Data is processed and worked upon to get information. Data may be of different types. It may be text, numeric or in decimal values. The different types of data are known as data types. Data types are an important factor in any programming language.

There is a need for different types of variables to store information, while writing a code in any programming language. Variables do nothing but just reserve memory to store values at some location in the computer. However, you are not required to know the exact location of the variable. You just need to know the two things about a variable that are name of the variable and the type of data it can store.

In this chapter, you will learn about the meaning and purpose of types. You will learn about type equivalence and type compatibility. You will also learn about composite types such as records, arrays, string and pointers.

3.2 Type Systems and Type Checking

Computer hardware interpret bits in a memory in instructions, addresses, characters, and integer and floating-point numbers. The bits are untyped, but the hardware on machines do not try to keep track of which locations are interpreted as instructions in memory. High-level languages always relate types with values to provide the contextual information and error checking.

Normally, a type system consists of following:

- a method to define the types and relate them with language constructs
- a set of rules for type equivalence, type compatibility, and type inference

The constructs who have types are those that have values or refer to objects that have values. These constructs include constants, variables, record fields, parameters, and subroutines, etc. Type equivalence is used to determine when the types of two values are same. Type compatibility is used to determine when the type can be used in a given context. Type inference is used to determine the type of an expression according to the constituent parts or surrounding context. A language that has polymorphic variables, it is necessary to differentiate between the type of a reference or pointer.

In some languages, subroutines are considered as types. Subroutines are types if they are of first- or second-class values. It means if they can be passed as parameters, returned by functions, or stored in variables. In these cases, there is a construct whose value is a dynamically determined subroutine. Type information allows the language to limit the acceptable values who provide a specific subroutine interface.

3.2.1 Polymorphism

The word poly means 'many' and morphism means 'form/state'. The ability to take more than one form is referred to as polymorphism. Let us first understand it with a real-life example, consider yourself as a person, who behaves like a student in school, behaves like a son/daughter to parents, behaves like a sister/brother to siblings, behaves like buddy to friends and behaves like a passenger in bus/train. It means, you are a single person but can take many forms.

In technical terms, polymorphism is the capability of using an operator or a function in different ways. A single function or an operator can function in many ways on different instances depending upon the usage.

Polymorphism is of two types:

- **Static polymorphism (compile time)**: Which function is to be called is decided at compile-time. It can be achieved through function overloading and operator overloading.
- **Dynamic polymorphism (runtime time)**: Which function is to be called is resolved at run-time. It can be achieved through function overriding.

- **Static polymorphism (compile time)**: Which function is to be called is decided at compile-time. It can be achieved through function overloading and operator overloading.
- **Dynamic polymorphism (runtime time)**: Which function is to be called is resolved at run-time. It can be achieved through function overriding.

In object-oriented languages, subtype polymorphism allows a variable X of type T to refer to any object of any type that is derived from T. These derived types should support the operations of base type; therefore, the compile can ensure that any operation that is acceptable for an object type T with be allowed for any object referred to by X.

Most languages such as C++, Eiffel, Java, and C# provide explicit parametric polymorphism. It allows the programmer to define classes with type parameters. Generics are used for container (collection) classes such as "list of T" (List<T>), "stack of T" (Stack<T>), etc. where T is unspecified. Smalltalk, Objective-C, Python, and Ruby use both parametric and subtype polymorphism.

3.2.2 Classification of Types

There is a need for different types of variables to store information, while writing a code in any programming language. Variables do nothing but just reserve memory to store values at some location in the computer. However, you are not required to know the exact location of the variable. You just need to know the following two things about a variable:

- Name of the variable
- The type of data it can store

All the variables are stored in the form of ones and zeros but interpreted in different ways based on the data type they store.

Data type basically defines the type of data that a variable can store. On defining the data type of a variable, the operating system assigns it some memory and defines what this reserved memory can store. Some of the commonly used data types are integer, character, floating point, wide character, double floating point, Boolean and so on.

Booleans or logicals are implemented as a single byte where 1 represents as true and 0 represents as false. In C language, there is not a Boolean type. C needs an integer where zero means false and anything else means true.

Characters are also implemented as one byte using the ASCII encoding. However, some languages such as Java and C# use two bytes to accommodate the Unicode character set. Unicode is an international standard which is used to capture the characters of a wide variety of languages. The first 128 characters \u0000 through \u007f are similar to ASCII.

3.2.3 Numerical Types

3.2.3 Numerical Types

Various languages such as C and Fortran use different lengths of integers and real numbers. However, some languages do not have different lengths. When there is a difference in precision across the language, then it led to a lack of portability. It means that the programs from one system may produce runtime error on another system. Languages such as Java and C# do not use several lengths of numeric types.

Languages such as C, C++, C# and Modula-2 use both signed and unsigned integers. Some languages such as Fortran, C99, Common Lisp, and Scheme use a built-in complex type that is implemented as a pair of floating-point numbers that are real and imaginary Cartesian coordinates. Languages such as Scheme and Common Lisp has built-in rational that has a pair of integers used to represent the numerator and denominator. Several languages support decimal types that use a base-10 encoding.

Integers, Booleans, and characters are examples of discrete types. Some other discrete types are the two varieties of user-defined types which are enumerations and subranges.

3.2.4 Enumeration Type

Enumerations were first introduced in Pascal. It helps in creating readable programs and helps the compiler to find programming errors. An enumeration type consists of named elements. In Pascal language, enum can be written as:

type month = (jan, feb, mar, apr, may, june, july, aug, sept, oct, nov, dec)

The values in an enumeration type are arranged, so it is possible to compare them such as jan < feb and you can also determine the predecessor or successor of an enumeration value such as tomorrow := succ(today)).

It also helps in writing the enumeration-controlled loops in the following manner:

for all_months := jan to dec

do begin ...

It is also used enumerations as index arrays:

var regular_month : array [month] of integer;

In C, the enumeration can be declared in the following manner:

In C, the enumeration can be declared in the following manner:

Enum enum_name {const1, const2, const3.....};

For example:

enum month {jan, feb, mar, apr, may, june, july, aug, sept, oct, nov, dec};

3.2.5 Subrange Types

Similar to enumerations, subranges were also introduced in Pascal. A subrange is a type whose values consists of a contiguous subset of the values of some discrete base type. In Pascal and other languages, subranges can be declared of integers, characters, enumerations, and other subranges. The following shows an example of subrange in Pascal:

type month = jan..dec;

day = mon..fri;

When explicit subranges are used, it helps in the documenting the program. A comment can also be used a documentation; however, comments can be outdated or can be omitted. Since, compiler analyses a subrange declaration, then it knows the expected range of subrange values, and can be used to generate code to perform dynamic semantic checks.

3.2.6 Composite Types

Non-scalar data types are known as composite or constructed types. These types are created by applying a type constructor to one or more simpler types. These types include records (structures), variant records (unions), arrays, sets, pointers, lists, and files. Let us discuss these types as follows:

- **Records (structures)**: They were introduced in Cobol and is supported by most languages. A record consists of fields where each field belong to a simple type. It is similar to a mathematical tuple or the Cartesian product.
- **Variant records**: Variant records are those records where only one field of the variant record would be valid at a given time. A variant record is the union of the field types instead of the Cartesian product.
- **Arrays**: Arrays are one of the most common composite types. It is similar to a function that is used map the members of an index type to the members of a component type. The array of characters is known as

- **Records (structures)**: They were introduced in Cobol and is supported by most languages. A record consists of fields where each field belong to a simple type. It is similar to a mathematical tuple or the Cartesian product.
- **Variant records**: Variant records are those records where only one field of the variant record would be valid at a given time. A variant record is the union of the field types instead of the Cartesian product.
- **Arrays**: Arrays are one of the most common composite types. It is similar to a function that is used map the members of an index type to the members of a component type. The array of characters is known as a string and can be used to perform special-purpose operations.
- **Sets**: Sets were also introduced in Pascal. A set is a type of a mathematical powerset of its base type. It is often discrete. A variable that is of set type contains a collection of different elements of the base type.
- **Pointers**: A pointer is a reference to an object of the base type of pointer. Pointers are used often but they are not implemented as addresses. They are mostly used as recursive data types. A type T is recursive if an object of type T consists of one of more references to other objects of type T.
- **Lists**: Lists also contains a sequence of elements, however there is no use of mapping or indexing. A list is usually defined recursively as either an empty list or a pair of a head element and reference to the list. In order to find a given element in the list, a program needs to examine all the elements.
- **Files**: Files are the data types that are used to represent data on mass-storage devices. Similar to arrays, files can be also treated as a function that is used to map members of an index type to members of a component type. Usually files have a current position that has an index that can be used in consecutive operations.

3.2.7 Type Checking

Normally, in statically typed languages, the definition of an object should always specify the object's type. In the following sections, you will learn about the type equivalence, type compatibility, and type inference. Type compatibility is used to determine when an object of a certain type can be used in a certain context. Type compatibility is used to determine if the objects and contexts are compatible even when their types are different. Type inference is used to determine types of the subexpressions. It is important in languages such as ML, Miranda, and Haskell, where the type information is inferred.

3.2.8 Type Equivalence

In a language where a user can define types, there are two principal methods to define the type equivalence – structure equivalence and name equivalence. Static equivalence is based on the content of type definitions. It means that two types are same if they consist of same components. It is used in languages such as Algol-68, Modula-3, C and ML. Name equivalence is based on the lexical occurrence. It means that each definition introduces a new type. It is used in languages such as Java, C#, standard Pascal, and Ada.

The structural equivalence depends on languages. To find out if the two types are structurally equivalent, then a compiler can expand its definitions by replacing the embedded type names with the definitions. Then it consists of a long string of type constructors, fieldnames, and built-in types. If these strings are same, then the types are equivalent, and conversely. In name equivalence, if the programmer writes two type definitions, then those definitions should mean different types.

3.2.9 Type Compatibility

Most of the languages do not need type equivalence. However, in these languages, the type of a value should be compatible with the context where it appears. For example, in an assignment statement, the type of the right hand should be compatible with the left hand. The type of the operands of + should be compatible with the type that supports addition such as integers, real numbers, or perhaps strings or sets.

The definition of type compatibility depends on languages. For example, Ada language uses a strict approach where an Ada type S will be compatible with an expected type T if:

- S and T are equivalent
- One of them is a subtype of the other
- Both are arrays that have same numbers and type of elements

When a language allows a value of one type is used in a context that needs another type, the language performs an automatic, implicit conversion to the expected type. This conversion is known as a type coercion. C has a weak type system and performs coercion. It allows value of numeric type to be mixed in expressions and coerce types back and forth. Example 1 shows you a code that shows coercion in C:

Example 1: Coercion in C

```
short int i;

unsigned long int u;

char c;      /* It can be signed or unsigned */

float f;     /* It is of single-precision */

double d;    /* It is of double-precision */

...
```

```
short int i;
unsigned long int u;
char c;      /* It can be signed or unsigned */
float f;     /* It is of single-precision */
double d;    /* It is of double-precision */
...
i = u;  /* the low order bits of l are used as a signed number. */
u = i;  /* s is extended to a longer length and then treated as an
           unsigned number. */
i = c;  /* c is interpreted as a signed number. */
f = u;  /* l is converted to floating-point and some precision may be
           lost.*/
```

```
f = u;  /* l is converted to floating-point and some precision may be
           lost.*/
d = f;  /* f is converted to the longer format, so no precision lost.*/
f = d;  /* d is converted to the shorter format, so precision may be
           lost. */
```

In modern languages, static typing is usually supported and not type coercion. However, coercions as used to support abstraction and program extensibility by using new types with the old ones. This helps in scripting languages. C++ provides a rich, programmer-extensible set of coercion rules. When a new type is defined in C++, the programmer can use coercion operations to convert values of the new type to and from existing types. They provide flexibility with the language but one of the most difficult C++ features to understand.

3.2.10 Type Inference

In previous sections, you have learned that how type checking ensures that the components of an expression have appropriate types. However, you need to find out the type of complete expression. The result of an arithmetic operator will be the same type as the operands. The result of a comparison is Boolean. The result of the function call has the type that is declared in the function's header. However, in cases of subranges and composite objects, they do not use the types of the operands.

3.3 Records and Variants

Records types are used to store heterogenous data. Some languages such as Algol 68, C, C++, and Common Lisp use the term structure which is declared with the keyword struct instead of record. Structures in C++ is a collection of variables of different data elements, i.e. different types of variables under the same name. The data elements will have different types and different lengths. Java has no notion of struct. C# uses a reference model for the variable of class types. C# structs do not support inheritance.

3.3.1 Records (Structure)

In C++, structures encapsulate data members that have different data types. For instance, you want to store complete information of a student viz. student name, class, section, roll number and school name. As you can see, each information is of different type and writing the code separately for them could make the program complex and difficult to understand if the information of multiple students is to be maintained. By defining a structure, we can keep the information of each student under a single unit i.e. within the structure.

The syntax of a structure in C++ is as follows:

```
struct structure_name
{
        type member_name1;
        type member_name2;
        .
```

```
        .
        .
        .
        type member_nameN;
    } object_names;
```

where:

- struct: It is the required keyword for declaring a structure.
- structure_name: It is the structure tag i.e. the name of the structure type.
- member_name: It specifies the name of the data member of any type, such as pointers, arrays or other structures.
- object_names: It represents the valid identifiers of objects of structure type.

In C language, a keyword struct is used to define a structure in the program. All members of the structure are enclosed within the curly braces and the closing curly braces need to terminate with a semicolon [;]. The syntax for defining a C structure is as follows:

```
struct structure_name
{
    data type structure member1;
```

```
        data type structure member1;

        data type structure member2;

        data type structure member3;

        …

    };
```

Where

- struct is a keyword for defining the structure.
- data type can be either int, float, char, and others.
- structure_name defines the name of the structure. The naming of structure should follow the same concept as the naming of the identifier.

Example 2 shows an example of a structure in C:

Example 2: Structure in C

```
struct person

{

        char[20] name;
```

```
        int age;

        char[5] gender;

    };
```

The lines of preceding code are defining a structure, person, which contains three members along with their data types. These members are the name, age, and gender.

The declaration of a structure in C is little different from the declaration of other data types. Since it is a user-defined data type, hence, the struct keyword is used for declaring a structure. The syntax for declaring a structure is as follows:

struct structure_name structure_var1, structure_var2,… ;

Where

- struct is a keyword for defining the structure
- structure_name defines the name of the structure.
- structure_var1, structure_var2,… are the structure variables

or

struct structure_name

{

…

…

} structure_var1, structure_var2;

Structure initialisation stores the values of each member of the structure into curly braces. The syntax for initialising a structure is as follows:

structure_variable = {value of member1, value of member 2,

...};

Where structure_variable contains the structure variables. For example, the statement “person1 = {“Ram”, 12, “M”}” is initialising values to one of the structure variable person1 of the structure person.

3.3.2 Accessing Structure Members

The C programming language uses a member access operator or dot operator [.] to access the specific element of a structure. The syntax of accessing a structure element is as follows:

structure_variable.structure_member

OR

structure_variable.structure_member = value

Example 3 shows a program where a structure “person” is defined that contains three structure members name, age and gender. Two structure variables “p1” and “p2” gets initialised with values. The printf() function prints each member of the structure variable using the dot operator one by one. The sizeof() operator gives the size of structure variable “p1”. The statement “person1.name” is accessing the name of the structure “person” whereas the “person1.age = 20” is storing value to the member age of the structure “person”.

Example 3: Accessing Structure Members

```
/* Structure program */

#include<stdio.h>

#include<conio.h>
```

```
struct person
{
      char name[10];
      int age;
      char gender[10];
};
void main()
{

       struct person p1 = {"Ram",18,"Male"}; // Structure declaration
 & initialization
       struct person p2 = {"Rita",17,"Female"};
       printf("\n Person1 \n ******** \n");
```

```
printf("\n Person1 \n ******** \n");
printf("\n Name = %s",p1.name);
printf("\n Age = %d",p1.age);
printf("\n Gender = %s",p1.gender);
printf("\n \n Person2 \n ******** \n");
printf("\n Name = %s",p2.name);
printf("\n Age = %d",p2.age);
printf("\n Gender = %s",p2.gender);
printf("\n ****************\n");
// Printing address and size of person1
printf("\n Size of structure member - p1 = %d", sizeof(p1));
printf("\n Address of member1= %u",p1.name);
```

```
        printf("\n Address of member1= %u",p1.age);

        printf("\n Address of member1= %u",p1.gender);

        getch();

    }
```

Figure 1 displays the output of Example 3 where information of two persons are printed. The sizeof() operator returns 22 [10+2+10 = 22 bytes] for the variable “p1”. The addresses of the structure members are also printed.

```
Person1
********

Name = Ram
Age = 18
Gender = Male

Person2
********

Name = Rita
```

```
Name = Rita
Age = 17
Gender = Female
***************

Size ofstructure member - p1 = 28
Address of member1= 2393129728
Address of member1= 18
Address of member1= 2393129744
```

Figure 1: Output of Example 3

3.3.3 Applications of a Structure

A structure is a useful data structure for doing programming efficiently. Structure has many applications. Here are some specific applications of structures.

Array of Structure

As we know that structure is used to store different type of information that are related to each other. For example, the information of the students in a class can be stored using a structure. However, one structure variable can store information of only one student. In case where you need to store the information of all the students of a class, you need to create separate structure variable for each student. This will be clumsy and time consuming. In such a situation, you can create array of the structure. This will help the developer reduce the lines of codes which he would have written in creating separate structure variables. The syntax for creating an array of structure is as follows:

struct structure_name structure_var[size] ;

Where

- struct is a keyword for defining the structure
- structure_name defines the name of the structure.
- structure_var[size] defines the array of structure where size contains the size of the array.

For example, the statement “struct person person1[20];” declares an array of 20 structure variables of the structure “person”. Example 4 shows a program that defines a structure “student” that contains two members name and age. An array of 20 structure is declared within the main() function. A for loop is used to read the values of each structure member for each structure variable.

Example 4: Array of Structure

```
/* Structure program */

#include<stdio.h>

#include<conio.h>

struct student

{

      char name[20];

      int age;
```

```
}; // Structure definition
void main()
{
      int i,n;
      struct student s1[10]; // array of structure
      printf("\n Enter number of student");
      scanf("%d",&n);
      for(i=1;i<=n;i++)
      {

            printf("\n Enter name and age of student[%d]: ",i);
            scanf("%s",s1[i].name);
```

```
            scanf("%d",&s1[i].age);
        }
        printf("\n Students information ******** \n");
        for(i=1;i<=n;i++)
        {
            printf("\n Name = %s",s1[i].name);
            printf("\t Age = %d",s1[i].age);
        }
    getch();
    }
```

Figure 2 displays the output of Example 4 where 3 is entered for the array size. Now, one by one, two values for each student is entered and printed:

```
Enter number of student3
```

```
Enter number of student3

Enter name and age of student[1]: John 18

Enter name and age of student[2]: Eva 16

Enter name and age of student[3]: Merlin 14

Students information ********

Name = John     Age = 18
Name = Eva      Age = 16
Name = Merlin   Age = 14
```

Figure 2: Output of Example 4

Nesting of Structure

The C programming language permits a user to use structure within another structure. This is called nesting of structure. The outer structure can access all the members of the inner structure. The following program defines this concept.

In Example 5, the program defines structure within a structure. The outer structure “student” contains an inner structure “marks” that contains two structure members. The outer structure variable “s1” is reading values for both structure members using the dot operator. M1 is the structure variable of the inner structure and s1.m1.subject1 is reading value.

Example 5: Nested Structure

```
/* Structure within structure */

#include<stdio.h>

#include<conio.h>

struct student

{

      char name[20]; // Outer structure

      int age;

      struct marks

      {

            int subject1; // Inner structure
```

```
        int subject2;
    } m1;
    }s1; // Structure definition
void main()
{
    strcpy(s1.name,"Raj"); // initializing the value of outer
                                structure member (name)
    s1.age = 18; // initializing the value of outer structure member
                    (age)
    s1.m1.subject1 = 80; // initializing the value of inner structure
                            member (subject2)
    s1.m1.subject2 = 90; // initializing the value of inner structure
```

```
                                member (subject2)

        printf("\n Students information ******** \n");


          printf("\n Name = %s",s1.name);

          printf("\n Age = %d",s1.age);

          printf("\n Subject1 = %d",s1.m1.subject1);

          printf("\n Subject2 = %d",s1.m1.subject2);

          getch();

    }
```

Figure 3 displays the output of Example 5:

```
Students information ********

Name = Raj
```

```
Name = Raj
Age = 18
Subject1 = 80
Subject2 = 90
```

Figure 3: Output of Example 5

Passing Structure to a Function

A function is a useful feature of programming languages. It reduces the number of lines of a code. Programmers can use structure and function together in different ways. For example, they can pass an entire structure to a function; can pass each member of a structure to a function. The following program is defining the concept where an entire structure is passed to the structure.

Example 6 shows a program that defines a structure “student” that contains two members, name and age. An array of 20 structures is declared within the main() function to accommodate the structure that is passed to the function “show()”. This show() function prints the values.

Example 6: Passing Structure to the Function

```
/* Passing structure to the function */

#include<stdio.h>

#include<conio.h>

struct student

{
```

```
{
      char name[20];

      int age;
}; // Structure definition
void show(struct student ss[], int n) // structure within function
{
      int i;
      printf("\n  Printing  students  information  within  function
      ******** \n");
      for(i=1;i<=n;i++)
      {
            printf("\n Name = %s",ss[i].name);
```

```
            printf("\t Age = %d",ss[i].age);
        }
    }
    void main()
    {
        int i,n;
        struct student s1[10]; // array of structure
        printf("\n Enter number of student");
        scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
            printf("\n Enter name and age of student[%d]: ",i);
```

```
            scanf("%s",s1[i].name);
            scanf("%d",&s1[i].age);
        }
        show(s1,n); // Function call
        getch();
    }
```

Figure 4 displays the output of Example 6 where 5 is entered for the array size. Now one by one, two values for each student is entered and is passed to the function show(). The function prints all the information of the 5 students:

```
Enter number of student5

Enter name and age of student[1]: John 16

Enter name and age of student[2]: Merlin 17

Enter name and age of student[3]: Eva 18
```

```
Enter name and age of student[3]: Eva 18

Enter name and age of student[4]: Ron 15

Enter name and age of student[5]: Jay 18

Printing students information within function ********

Name = John      Age = 16
Name = Merlin    Age = 17
Name = Eva       Age = 18
Name = Ron       Age = 15
Name = Jay       Age = 18
```

Figure 4: Output of Example 6

3.3.4 Variants (Union)

Union is also a user-defined data type that supports storing different kinds of data like structure. The union is a heterogeneous data structure for storing different numbers, strings, and real numbers in one place. It is similar to the structure but there is one major difference. The difference is that all members of the union use the same location for storage whereas the members of a structure have their own storage. It indicates that union can handle one member at a time.

C language provides a keyword “union” for defining the union in a program. All members of the union are enclosed within curly braces and the closing curly braces should terminate with a semicolon [;]. The syntax for defining the union is as follows:

```
union union_name

{

data type union member1;

data type union member2;

data type union member3;

...

};
```

Where

- union is a keyword for defining the union.
- data type can be either int, float, char, and others.
- union_name defines the name of the union. The naming of union should follow the same concept as that of naming of the identifiers.

Let us take an example:

```
union employee

{

char[20] name;

int age;

float salary;
```

```
float salary;

};
```

These preceding example defines a union “employee” that contains three members. These members are the name, age and salary.

The “union” keyword is used to declare a union. The syntax for declaring a union is as follows:

```
union union_name union_var1, union_var2,... ;
```

Where

- union is a keyword for defining the union
- union_name defines the name of the union.
- union_var1, union _var2,... are the union variables.

OR

```
union union_name

{

....

.....

} union_var1, union_var2;
```

For example, the statement “union employee emp1, emp2;” declares two variables for the union “employee”, which is defined above. Each union variable contains the values of the structure employee.

3.3.5 Union Initialisation

Union initialisation in the same way as done for a structure. The syntax of the union initialisation is as follows:

3.3.5 Union Initialisation

Union initialisation in the same way as done for a structure. The syntax of the union initialisation is as follow:

union_variable = { value of member1,value of member 2, ...};

Where union_variable contains the union variables.

For example, the statement “emp1 = {“Ram”, 12, 10000.0}” initialises values to one of the union variable “emp1” of the structure “employee”.

3.3.6 Accessing Union Members

Union also uses the member access operator or dot operator [.] to access an element. The syntax for accessing the union element is as follows:

union_variable.union_member

OR

union_variable.union_member = value

For example, the statement “emp1.name” is accessing the name of the union “employee” whereas the “emp1.age = 20” is storing value to the member age of the union “employee”.

Example 7 shows a program that defines the union “employee” that contains three members name, age and salary:

Example 7: Union Program

```
/* Union program */

#include<stdio.h>
```

```
#include<conio.h>

union employee

{

      char ename[100];

      int eage;

      float esalary;


  };

  void main()

  {

        union employee emp1,emp2; // Declaring the union variables

        // Printing address and size of employee1
```

```
  printf("\n Size of union member - emp1 = %d", sizeof(emp1));

  printf("\n Enter values for union");

  printf("\n Enter name:");

  scanf("%s",emp1.ename);

  printf("\n Enter age:");
scanf("%d",&emp1.eage);

printf("\n Enter salary:");

scanf("%f",&emp1.esalary);

strcpy(emp2.ename,"Sohan");

emp2.eage = 24;

emp2.esalary = 7500.0;

printf("\n Name = %s",emp1.ename);
```

```
        printf("\t Age = %d",emp1.eage);

        printf("\t Salary = %f",emp1.esalary);

        printf("\n Name = %s",emp2.ename);

        printf("\t Age = %d",emp2.eage);

        printf("\t Salary = %f",emp2.esalary);

        getch();

    }
```

Figure 5 displays the output of Example 7 where the sizeof() operator returns 100 as it is the largest memory address declared by union member “ename”. The output is not displaying the exact values, which are initialised to the union members, and only one member is printed properly. It is because the union can access only one member at one time:

```
Size of union member - emp1 = 100
Enter values for union
Enter name:John

Enter age:40
```

Figure 5 displays the output of Example 7 where the sizeof() operator returns 100 as it is the largest memory address declared by union member "ename". The output is not displaying the exact values, which are initialised to the union members, and only one member is printed properly. It is because the union can access only one member at one time:

```
Size of union member - emp1 = 100
Enter values for union
Enter name:John

Enter age:40

Enter salary:40000

Name =          Age = 1193033728        Salary = 40000.000000
Name =          Age = 1172987904        Salary = 7500.000000
```

Figure 5: Output of Example 7

3.4 Arrays and Strings

Arrays are one of the most common composite data types. They are homogeneous. They are similar to a mapping from an index type to a component or element type. In many languages, a string is simply an array of characters. In other languages, strings have special status, with operations that are not available for arrays of other sorts. Let us discuss them.

3.4.1 Array

An array is a type of data structure that can hold multiple values of similar type of data. According to the technical definition of the array, “Array is a homogeneous data structure that sequentially stores the same type of data and performs different types of operations on these data”.

In C, an array is a derived data type for storing similar type of data. The primary data types are integer, float and char. These data types store numbers, real numbers and characters, respectively. An array stores numbers using the integer data type, real number using float and string using char. Due to this, an array is known as the derived data type.

An array has many features or characteristics that make it important for programming. Here are some major features of an array:

- Array can store similar data type. Each data item of an array is called the array element. For example, if the data type of the array is char then it only stores the character data or string.
- Every programming language represents the array using a subscript written within square brackets“[ ]”. For example, “student[ ]” represents an array where the name of the array is the student. The value within the subscript operator defines the size of an array. In other words, student[60] can store the data of 60 students. The value within subscript is also called index or subscript variable.
- Computer stores each data element continuously in the memory. For example, if there are 5 students in an array “Student” then the computer will store it in the memory as:

  Student[0] = Student1

  Student[1] = Student2

  Student[2] = Student3

  Student[3] = Student4

  Student[4] = Student5

Student[4] = Student5

Here, Student[0],...., Student[4] are the memory addresses of the computer where the computer stores the actual data. The lowest address student[0] stores the first element whereas the highest address means student[4] stores the last element of the array.

3.4.2 Array Declaration

The declaration of an array in C follows the same concept as the declaration of variables. While declaring an array, its data type needs to be specified. The naming of array should follow the same concept as the naming of the variables. The major rules of naming of an array are as follows:

- The name of the array should start with a character and can include underscore or digit.
- The array name should not include any keyword.
- Special symbols should not be used in the array name.

The syntax for declaring an array is as follows:

data type array_name[arraysize]

Where

- data type can be either int, float, char, and others.
- array_name defines the name of the array.
- arraysize contains any positive numeric value that defines the size of the array.

Example 8 shows a program that declares three types of array including integer, float and character. The sizeof() operator returns the size of each array.

Example 8: Array Declaration

```
/* Array declaration */
```

```
/* Array declaration */
#include<stdio.h>
void main()
{
      int a[4]; // Integer array with size 4
      float f[5]; // Float array with size 5
      char ch[10]; // Character array with size 10
      printf("\n Memory size in computer");
      printf("\n *************************");
      printf("\n Size of Integer array = %d", sizeof(a));
      printf("\n Size of float array = %d", sizeof(f));
      printf("\n Size of character array = %d", sizeof(ch));
```

```
        printf("\n Size of character array = %d", sizeof(ch));

        getch();

    }
```

Figure 6 displays the output of Example 8.

```
Memory size in computer
***********************
Size of Integer array = 16
Size of float array = 20
Size of character array = 10
```

Figure 6: Output of Example 8

3.4.3 Array Initialisation

Initialisation assigns values to the array elements. A programmer can assign values to arrays at either run time or at compile time. C language provides the feature to assign values of the array elements at compile time and run time. The following subsection defines both methods of assigning values to the array elements.

Initialisation assigns values to the array elements. A programmer can assign values to arrays at either run time or at compile time. C language provides the feature to assign values of the array elements at compile time and run time. The following subsection defines both methods of assigning values to the array elements.

Initialisation at Compile Time

Compile time initialisation is a type of static binding where values are stored in the program itself. The curly brackets {} are used to initialise the values in the array. The syntax of such initialisation is as follows:

data type arrayname[ ] = { value1,value2, ...} or { values}

Where values should be according to the size of the array.

Initialisation at Run Time

Run time initialisation is a type of dynamic binding where values are assigned explicitly after execution. The programmer uses scanf() functions for reading values at run time. The syntax of such an initialisation is as follows:

```
data type arrayname[ size];

scanf("%d %d...%d", &arrayname[0], &arrayname[1],...,&arrayname[size-1])

OR

for(i =0 ; i < size; i++)

{

scanf("%d", &arrayname[i]);

}
```

3.4.4 Accessing Array Elements

3.4.4 Accessing Array Elements

Array elements are accessed by using index or subscript of the array. Here, index or subscript is a numeric value, which defines the size of the array. The syntax of such initialisation is as follows:

data type arrayname[size] = { value1, value2,....}

arrayname[0] = value1;

arrayname[1] = value2;

Let us see an array sample that contains 5 values as follows:

int sample[5] = { 10,20,30,40,50}

then sample[0] will access the first element means 10

In the same way,

sample[1] = 20 [Accessing the second element]

sample[2] = 30 [Accessing the third element]

sample[3] = 40 [Accessing the fourth element]

sample[4] = 20 [Accessing the fifth element]

3.4.5 Strings

An integer array stores a number of integers, in the same way, a character array stores a group of characters. This group of characters or the character array is nothing but a string.

In C, a string is used as variables or characters. A string variable can change the values; however, a string constant contains some fixed values. In simple words, a string constant is a one-dimensional array, which is

In C, a string is used as variables or characters. A string variable can change the values; however, a string constant contains some fixed values. In simple words, a string constant is a one-dimensional array, which is terminated by a null character ['\0']. For example, char surname [] = {'a', 'b', 'c', 'd', '\0'} is declared as string.

A string constant is often called null terminated character array. In C programming language, if a string is not terminated by the null character then it is not a string but it is only an array of characters.

Each character in a string occupies one byte of memory and the last character of the string is always '\0'. For example, if the first character is stored in address 1001 then the next character will be stored in 1002, and so on. A string can contain other control or special characters such as \n, \t, or others.

In programming, '\0' and '0' are different from each other. Here, '\0' is a character whose ASCII [American Standard Code for Information Interchange] value is 0 whereas '0' is number or digit whose ASCII1 value is 48.

3.4.6 String Declaration

Like the declaration of an integer array, the declaration of the string follows the same rules. C programming language does not provide a specific data type 'string'; hence, the datatype char is used to declare a string or character array. The name of string variable should have to follow the same rules for the naming of variables, which is discussed in the previous chapters. In addition, declaration of the null character '\0' is not necessary in C programming as the C compiler automatically inserts it at the end of a string.

The syntax for declaring a string is as follows:

char string_name[stringsize]

Where

- char defines the data type
- string_name defines the name of the string.
- stringsize contains any positive numeric value that defines the size of the string.

The statement, char person_name[10] declares a string or character array where the size of the string is 10. It will store any 9 characters that would be stored from memory location 0 to 9.

3.4.7 String Initialisation

3.4.7 String Initialisation

String initialisation assigns values to the string variables. Like simple array initialisation, the programmer can assign values to string at either compile time or run time. The { } (parenthesis) initialises the string at compile time whereas the scanf() function assigns values to string at run time. The syntax for both the initialisations are as follows:

char stringname[ ] = { ‘value1’, ‘value2’, …, ‘\0’} or “values”

Where

- stringname contains the name of the string
- values contains the texts, words, or characters

OR

char stringname[size];

scanf(“%s”, stringname)

Where stringname contains the name of the string.

3.4.8 Operations on Strings

Arithmetical and logical operators perform different arithmetic and logical operations on variables. Similarly, different operations are possible on strings. These operations include determining the length of a string, concatenating strings, comparison of two strings, extracting parts of strings, string copy, and others. Programmers can either perform these operations manually by writing codes or can use the built-in string functions, which are available in the header file “string.h”.

The string.h header file contains many functions. The programmer can use these functions according to their programming needs. Programmer can check the available functions in the header file from the Help menu within the compiler.

Table 1 describes some major string functions:

Table 1: String Functions and Examples

String Functions	Description	Example
`strlen(stringname)`	The function returns the length of the string. The null character is not included in the length of the string.	`strlen("Computer")` will return 8.
`strlwr(stringname)`	The function converts the given string into lower case letters.	`strlwr("Computer")` will return "computer" in output.
`strupr(stringname)`	The function converts the given string into uppercase letters.	`strupr("Computer")` will return "COMPUTER" in output.
`strcat(string1`	The function concatenates to	`strcat("Demo"`

	letters.	"computer" in output.
strupr(stringname)	The function converts the given string into uppercase letters.	strupr("Computer") will return "COMPUTER" in output.
strcat(string1, string2)	The function concatenates to the string2 with the string1.	strcat("Demo", "string") will returns "Demostring" in output.

3.5 Pointers

Pointer is a variable that holds the address of another variable of the same data type. From this address, the pointer can directly access the value, which is stored in that address.

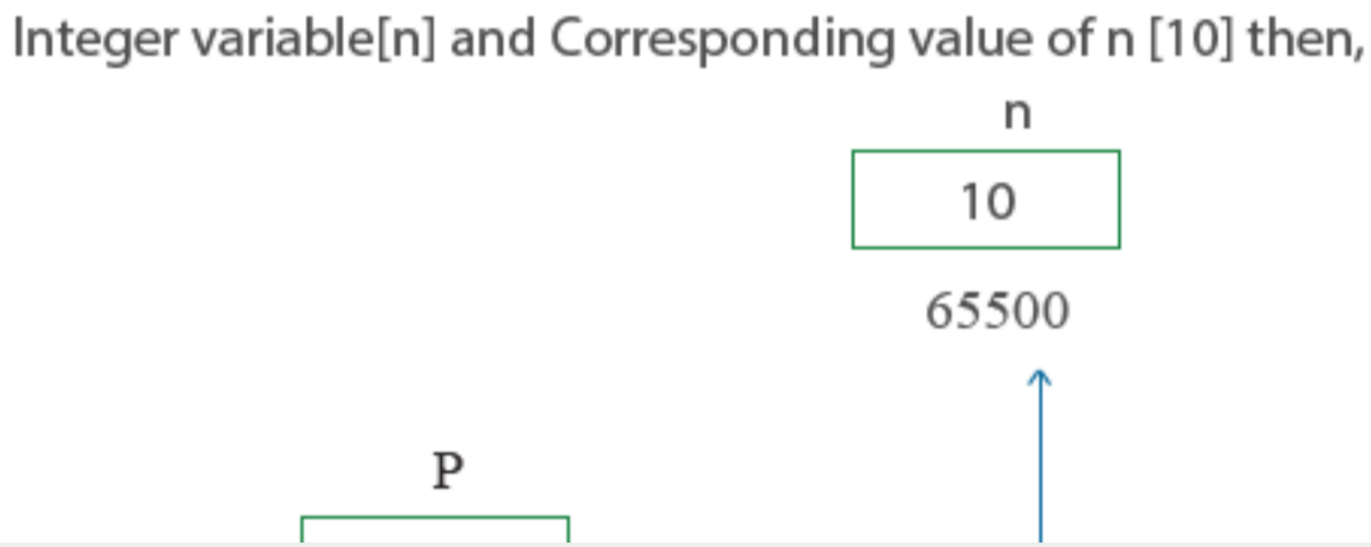
Consider the following variable declaration:

int n=10;

This declaration instructs the compiler to do the following three things:

- Reserve a space in the computer memory to hold an integer.
- Name the memory location as n.
- Store the value 10 in the location.

This declaration is represented in Figure 7:



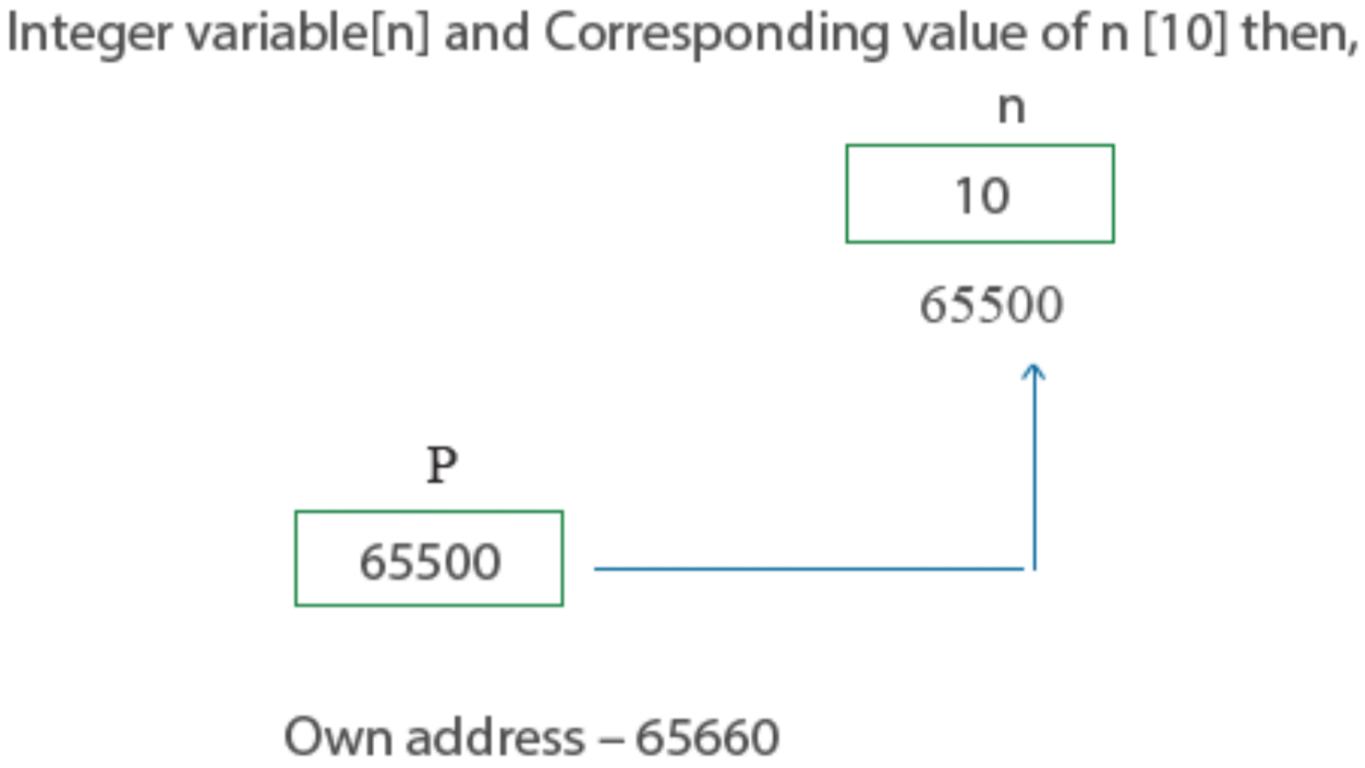


Figure 7: Memory representation of a Variable

As shown in Figure 7, n is a variable for which the compiler has allocated a space at memory location 65500. The value of the variable, 10, is stored at that memory location.

Consider the following declaration:

int *P=&n; \\Pointer declaration is explained in the subsequent section

A pointer variable is a special type of variable that can hold the address of another variable. In the preceding declaration, a pointer variable P is declared. It is initialised with the address of the variable, n. The pointer variable, P, itself has a memory location, which is 65660. On technical terms, it is called as pointer P points to the location 65500.

3.5.1 Pointer Declaration

Pointer variables in C are declared by using the asterisk operator [*]. The naming of pointer variable should follow the naming rules of identifiers as discussed in previous chapters. Pointer variable supports all data types. The syntax for declaring a pointer variable is as follows:

data type *pointer_name;

OR

data type* pointer_name;

Where

- data type can be int, float, char, void, and others.
- pointer_name defines the name of the pointer.

3.5.2 Initialisation of a Pointer Variable

The initialisation of a pointer variable assigns the address of other variables and for this it uses an address of operator [&]. This process is also called the referencing as it stores the address or reference of another variable. The syntax for initialising a pointer is as follows:

data type var_name;

data type *ptr_var;

ptr_var = &var_name // referencing or storing address of another variable

Where

- data type can be int, float, char, void, and others.
- var_name defines the name of the variable.
- ptr_name defines the name of pointer variable

3.5.3 Accessing Values Using a Pointer Variable

Pointer variables can also access the value of the specific variables whose address is referenced to that pointer variable. For this, the same unary operator [*] is used. This operator is also called the indirection operator and the process of accessing value of the variable whose address is stored in the pointer is called dereferencing. The syntax for accessing value using a pointer is as follows:

data type var_name;

data type *ptr_var;

ptr_var = &var_name // storing address of another variable

var_name = *ptr_name // dereferencing or accessing value of pointer variable

Where

- data type can be either int, float, char, void, and others.
- var_name defines the name of the variable.
- ptr_name defines the name of pointer variable

Table 2 describes the examples of pointer initialisation and accessing values using pointer variables:

Table 2: Examples of Pointer Initialisation

Examples	Explanation

Examples	Explanation

`int a, b;`
`int *p;`
`a = 21;`
`p = &a; // referencing`
`b = *p;`
	Declares two integer variables "a" and "b" and one integer pointer [*p]. The value 21 is assigned to variable "a". The integer pointer is storing the address of variable "a". It means the value of variable "a" is accessible using pointer. The line "b = *p" is accessing and storing the value of "a" into the variable "b".

`float fa, fb, *fp;`
`fa = 23.3;`
`fb = *&fa;`
	Declares two float variable [fa, fb] and one float variable [fp]. The value 23.3 is assigned to the variable "fa". The line "fb = *&fa" is first storing address and then accessing the value of variable "fa" and fb is assigned the value 23.3.

`int a, *pa = &a;`	Declares a variable "a" and initializing an integer pointer "*pa", which is storing the address of the variable "a".

`int a, b, c, *pd,d,s,t;`
`pd = &a;`
`d = *pd`
`pd = &b;`
`s = *pd;`
`pd = &c;`
`t = *pd;`
	Declares some integer variables [a,b,c,d,s,t] and one integer pointer [*pd]. The same pointer is storing the address of three variables and accessing the value of that address one by one.

3.5.4 Pointer Arithmetic

A pointer is a distinct feature of the C language that stores the addresses of other variables. C language permits the use of arithmetic operators with pointers. Pointers do not support all operators such as multiplication and division. They support increment/decrement operators, addition and subtraction operators. In addition, pointers are used with the relational operators, which means the comparison is possible between pointers. Programmer can write an expression using pointers and operators. They can use them in the following ways:

```
data type *ptr_var

ptr_var++ or ++ptr_var //Points to next integer after var
```

```
ptr_var++ or ++ptr_var //Points to next integer after var

ptr_var- - or - -ptr_var //Points to previous integer after var

ptr_var + number //Points to the next number after var

ptr_var - number //Points to the previous number after var

++*ptr_var //Will increment var by 1

--*ptr_var //Will decrement var by 1

ptr_a < ptr_b //Returns true only if a is stored before b

ptr_a > ptr_b //Returns true if a is stored after b

ptr_a == ptr_b //Returns true only if ptr_a and ptr_b point to same
data element.

ptr_a!= ptr_b //Returns true if ptr_a and ptr_b point to different
data element but of the same type.
```

```
ptr_a!= ptr_b //Returns true if ptr_a and ptr_b point to different
data element but of the same type.
```

In Example 9, *p1 and *p2 are two pointer variables and a and b are two integers whose values has been assigned. In this program, various pointer arithmetic operations are performed whose explanation is given in the output section.

Example 9: Pointer Operation

```
/*Pointer Operation */

#include <stdio.h>

int main()

{

      int a, b, *p1, *p2, x, y, z;

      a = 10;

      b = 5;

      p1 = &a;
```

```
p1 = &a;

p2 = &b;

x = *p1 * *p2 - 6;

y = 4* - *p2 / *p1 + 10;
```

```
printf("Address of a = %p \n", p1);

printf("Address of b = %p \n", p2);

printf("\n");

printf("a = %d, b = %d\n", a, b);
```

```
        printf("x = %d, y = %d\n", x, y);

        *p2 = *p2 + 3;

        *p1 = *p2 - 5;

        z = *p1 * *p2 - 6;

        printf("\na = %d, b = %d", a, b);

        printf("z = %d\n", z);

    }
```

The output of the Example 9 is given below. In this program, 'a' and 'b' are assigned with some values as 10 and 5 respectively. Address of a and b is stored in ptr1 and ptr2, respectively.

In the first operation, 'x' stores the product of the value of the specified addresses, i.e. 'a' and 'b' and then subtracts 6 from that. In the next statement, 'y' stores the operation which is followed from left to right as per its associativity and generates result as 8.

The printf() statements in the next lines are printing the memory address of the variables 'a' and 'b' where they are stored and 'i' depends on compiler to generate the memory address.

At the end, the values of all the variables are printed and displayed. Again, here in the next statement the value of p2 is incremented by 3 and stored in p2 and the value of p1 is assigned value as p2-5. In the next operation, variable 'z' stores the value of product p1 and p2-6 which gives 18.

```
Address of a = 0x7ffda7739dac
Address of b = 0x7ffda7739db0
```

```
a = 10, b = 5
x = 44, y = 8

a = 3, b = 8z = 18
```

Figure 8: Output of Example 9

3.5.5 Pointer and Arrays

Pointer and array are closely interrelated programming concept in C. An array stores a list of the similar data and uses indexing for accessing each element. The compiler allocates contiguous memory for an array. If there is an array “a” then “a” and “&a[0]” both are same that stores the base address or the address of the first element of the array. The remaining elements stored sequentially after this. It indicates that the array represents a constant pointer whose memory location does not change.

Pointer also follows this concept and stores the address of an element. The only difference between the pointer and array is that an array is a constant pointer unlike pointers. For printing and storing the array values, the concept of looping is used. For example, if a is an array then a[i] prints each element of the array. This a[i] converts into a pointer expression that returns the respective value of the array. Hence, a[i] is similar to *(a+i).

Whenever a pointer is used to access an array element then it moves into the computer memory according to the data type. For example, if there is an integer array, and a pointer is used for accessing elements then in each loop the pointer will move further by two-memory locations. This is because integer data type takes 2 bytes in memory. For float data type, it will move to the four-memory locations.

In simple words, the “a” and “*a” are equivalents to the “&a[0]” and “a[0]” respectively. In the same way, “a+i” and “*(a+i)” are equivalents to the “&a[i]” and “a[i]” respectively.

Example 10 initialises an array with five values and is declares one integer pointer [*p]. The pointer stores the base address of the array and prints all values of the array:

Example 10: Pointer and Array

```
/* Pointer and array */

#include<stdio.h>

#include<conio.h>

void main()

{

      int a[] = {10,20,30,40,50};

      int *p,i;

      printf("Pointer and Array");

      printf("\n Base address of array = %u \n ",a);

      p = &a[0]; // OR p = a; Storing base address


     printf("\n*********\n");
```

```
printf("\n Printing values of array using pointer\n");

for(i=0;i<5;i++)

{

      //printf("a[i] = %d\t",a[i]);

      //printf("%d\n",i[a]); /* All these lines are valid and
printing the same output*/

      //printf("%d\n",*(a+i));

      //printf("%d\n",*(p+i));

      //printf("%d\n",p[i]);

      printf("Address = %u \t ",p);

      printf("a[%d] = %d \n ",i,*p++);

} // End of for loop
```

```
getch();

}
```

Figure 9 displays the output of Example 10 where, at first, the base address of the array is printed [65516] which is assigned to the integer pointer. After that “p” and “*p++” are printing the address and the corresponding values of the array. For example, the first address is 65516 then the integer pointer automatically moves further by two memory location and prints the address 65518.

```
Pointer and Array
 Base address of array = 1476505920

*********

 Printing values of array using pointer
Address = 1476505920     a[0] = 10
 Address = 1476505924    a[1] = 20
 Address = 1476505928    a[2] = 30
 Address = 1476505932    a[3] = 40
 Address = 1476505936    a[4] = 50
```

3.5.6 Array of Pointers

Character pointer is pointing to the character array and prints one character at a time. Sometimes the users also need to print list of arrays such as array of strings. For using, array of stings, 2D array is used in which it is necessary to provide column size. In such a case, C programming language provides the feature to programmers to create and implement array of pointers. The array of character pointer supports the string of varying length.

The syntax for declaring an array of pointer is as follows:

data type *ptr_var[size];

Where

- data type defines any data type.
- ptr_var defines the name of the pointer.

3.5.7 Pointer and Functions

C language permits a programmer to use pointers with function in a different way. Both, pointers and functions reduce the lines of codes in C programs and also provide other advantages. When a single program uses both features together then definitely it increases the execution speed. The programmers can use pointers and functions in the following three ways:

1. They can pass the pointers to a function as an argument [call by reference]. In this method, an address of the variable, i.e., the pointer is passed to the function call, which can be further modified in the function definition.
2. In the second method, the function can return the pointer like other data types. It can be used as -“data type* function_name()”.
3. In the third method, the programmer can design pointer to function, which is a function pointer. This function pointer can store the address of other functions and call them using pointers. The statement, “data type (*function_name) ()” defines the function pointer.

3.5.8 Equality Testing and Assignment

2. In the second method, the function can return the pointer like other data types. It can be used as - data type* function_name() .
3. In the third method, the programmer can design pointer to function, which is a function pointer. This function pointer can store the address of other functions and call them using pointers. The statement, “data type (*function_name) ()” defines the function pointer.

3.5.8 Equality Testing and Assignment

For primitive data types such as integers, floating-point numbers, or characters, equality testing and assignment is simple with obvious semantics and implementations. For abstract data types, it can cause problems with both semantic and implementation.

Suppose let us take an example of comparing two character strings. The expression s=t is used to determine whether:

- They are aliases for one another.
- They occupy storage that is bit-wise identical.
- They contain the same sequence of characters.
- They would appear the same if printed.

It is possible that it is not possible to compare because of the garbage in currently unused portions of the space reserved for a string. In various cases, equality focuses on the differences between l-values and r-values when references are there. They are considered equal if they refer to the same object, or also if the objects to which they refer are in some sense equal. The first comparison is known as a shallow comparison and second comparison is known as a deep comparison.

In imperative languages, assignment operations can be both deep and shallow. A shallow assignment a:=b makes a refer to the objects that b is referring. A deep assignment will create a copy of the object to which b refers and make a refer to the copy.

3.6 Recap and Questions

3.6.1 Recap

Let us quickly recapitulate the following discussion points:

- A type system consists of a set of built-in types, a mechanism to define new types, and rules for type equivalence, type compatibility, and type inference.
- Type equivalence determines when two names or values have the same type.
- Type compatibility determines when a value of one type may be used in a context that “expects” another type.
- Type inference determines the type of an expression based on the types of its components or the surrounding context.
- A structure is a user-defined data type that supports different data types for storing different kinds of data. C language provides a keyword “struct” for defining the structure in the program.
- The C programming language uses a member access operator or dot operator [.] to access the particular element of the structure.
- Union is a user-defined data type that supports different data types for storing different kinds of data like structure.
- An array is the type of homogeneous data structure that stores the same type of data.
- A character array stores a group of characters. This character array is a string.
- A string constant is a one-dimensional array, which is terminated by a null character [‘\0’].
- The pointer is a variable that holds the address of another variable of the same data type.
- Pointers can directly access the memory that reduces the storage space of the computer memory.

3.6.2 Test Your Skills - Multiple Choice Questions

1. Which of the following data types consists of fields where each field belong to a simple type?

a. Records
b. Variants
c. Arrays

1. Which of the following data types consists of fields where each field belong to a simple type?

a. Records
b. Variants
c. Arrays
d. Sets

2. Which of the following data types are those records where only one field of the variant record would be valid at a given time?

a. Variant records (unions)
b. Arrays
c. Sets
d. Pointers

3. Which of the following data types is similar to a function that is used map the members of an index type to the members of a component type?

a. Variant records (unions)
b. Arrays
c. Sets
d. Pointers

4. Which of the following data types is a type of a mathematical powerset of its base type?

a. Variant records (unions)
b. Arrays
c. Sets
d. Pointers

5. Which of the following data types are used to represent data on mass-storage devices?

a. Variant records (unions)
b. Arrays
c. Files
d. Pointers

b. Arrays
c. Files
d. Pointers

3.6.3 Fill in the Blanks

1. _____ is used to determine when the types of two values are same.
2. _____ is used to determine when the type can be used in a given context.
3. _____ is used to determine the type of an expression according to the constituent parts or surrounding context.
4. _____ is the capability of using an operator or a function in different ways.
5. Most languages such as C++, Eiffel, Java, and C# provide explicit _____ polymorphism.

3.6.4 Theoretical Questions

1. What is the difference between type equivalence and type compatibility?
2. Discuss the structural and name equivalence for types. Name three languages that use each approach.
3. How can we use array within a structure? Explain with the help of an example.
4. What is difference between structure and union? How can we declare and initialise a union?
5. Explain the relation between an array and a pointer.

Chapter 4 : Subroutine and Control Abstraction

Learning Objectives

At the end of this module, you will be able to:

- Explain stack layout.
- Discuss calling sequences and passing parameters.
- Explain generic subroutines and modules.
- Explain exception handling.
- Discuss coroutines and events.

4.1 Introduction

A subroutine is a principal mechanism for control abstraction in most programming languages. It performs the operation on behalf of a caller who waits for the subroutine to finish before continuing execution. Most subroutines are parameterised, which means that the caller passes the arguments that affect the behaviour of subroutine. Arguments are known as actual parameters and they are mapped to the subroutine's formal parameters at the time. A subroutine that returns a value is known as a function and that does not return a value is known as a procedure.

In this chapter, the layout of the stack is reviewed and then the calling sequences are discussed that are used to maintain the layout. So, here you will learn about the subroutine parameters, the generic subroutines and modules that support explicit parametric polymorphism. You will also learn about the exceptional handling. And, at the end of this module you

At the end of this module, you will be able to:

- Explain stack layout.
- Discuss calling sequences and passing parameters.
- Explain generic subroutines and modules.
- Explain exception handling.
- Discuss coroutines and events.

4.1 Introduction

A subroutine is a principal mechanism for control abstraction in most programming languages. It performs the operation on behalf of a caller who waits for the subroutine to finish before continuing execution. Most subroutines are parameterised, which means that the caller passes the arguments that affect the behaviour of subroutine. Arguments are known as actual parameters and they are mapped to the subroutine's formal parameters at the time. A subroutine that returns a value is known as a function and that does not return a value is known as a procedure.

In this chapter, the layout of the stack is reviewed and then the calling sequences are discussed that are used to maintain the layout. So, here you will learn about the subroutine parameters, the generic subroutines and modules that support explicit parametric polymorphism. You will also learn about the exceptional handling. And, at the end of this module you will learn about coroutines and events as well.

4.2 Exploring Stack Layout

A routine is provided with a new stack frame, or activation record, at the top of the stack. This frame consists of arguments, return values, bookkeeping information, local variables, and/or temporaries. When a subroutine is returned, its frame is popped from the stack.

At a point of time, the stack point register consists of the address of either the last used location at the top of the stack or the first unused location. The frame pointer register consists of an address within the frame. Objects in a frame are accessed using the displacement addressing with respect to the frame pointer. If the size of an object is not provided at a compile time, then the object is placed in a variable-size area at the top of the frame. Its address and descriptor are stored in the fixed-size portion of the frame. If there are no variable-size objects, then every object in a frame has a statically known offset from the stack pointer. If the size of an argument is not known at the compile time, then that argument can be placed in a variable-size portion of the frame below the other arguments.

Languages that have nested subroutines and static scoping, such as Pascal, Ada, ML, Common Lisp, or Scheme, have objects that are located in the subroutines and are neither local nor global. These objects can be found by maintaining a static chain. Each stack frame consists of the reference to the frame of the lexically surrounding subroutine. This reference is known as static link. The saved value of the frame pointer that is restored on a subroutine return is known as a dynamic link.

4.3 Calling Sequences and Passing Parameters

To maintain the subroutine call stack, the calling sequence is used. It is a code executed by the caller immediately before and after a sub-routine call. Sometimes, calling sequence is also used to refer the combined operations of the caller — the prologue and the epilogue. Prologue is the code executed at the beginning of a routine and epilogue is the code executed at the end of the routine.

Some of the tasks that should be completed on the way into a subroutine are listed below:

- Passing parameters
- Saving the return address
- Changing the program counter
- Changing the stack pointer to allocate space
- Saving registers that contain important values
- Changing the frame pointer to refer to the new frame
- Executing initialisation code for any objects.

And, some of the tasks that should be completed on the way out to a subroutine are:

- Passing return parameters or function values
- Executing finalisation code for any local objects that require it
- Deallocating the stack frame
- Restoring other saved registers
- Restoring the program counter

The saving and restoring of registers are essential during subroutine and calls since they determine the state of the programming environment. Hence, saving registers is the trickiest division of labour. Ideally to avoid unnecessary overhead only those registers that are both in use in the caller and needed for other purposes in the callee should be precisely saved.

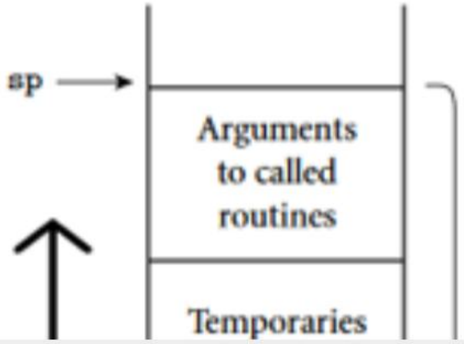
division of labour. Ideally to avoid unnecessary overhead only those registers that are both in use in the caller and needed for other purposes in the callee should be precisely saved. However, because of separate compilation it is difficult to determine these overlapping common registers. A simpler solution is for the caller to save all registers that are in use or for the callee to save all registers that it will overwrite.

Generally, the registers not reserved for any special purposes are divided into two sets of approximately equal size. One set is the caller's responsibility and the other is the callee's responsibility. A callee can assume that there is nothing of value in any of the registers in the **caller-saves** set and a caller can assume that no callee will destroy the contents of any registers in the **callee-saves** set. In the interests of code size, the compiler uses the callee-saves registers for local variables and other long-lived values, whenever possible. It uses the caller-saves set for transient values, which are less likely to be needed across calls. The result of these conventions is that the caller-saves registers are seldom saved by either party — the callee knows that they are the caller's responsibility and they don't contain anything important.

In languages that use nested subroutines, some part to maintain the static chain is performed by the caller instead of callee because this work depends on the lexical nesting depth of the caller. This approach is used for the caller to compute with the callee's static link. Two subcases that happen in this case are:

1. The callee is nested into the caller. In this case the callee's static link refer to the caller's frame.
2. The callee is k≥0 scopes "outward" which is closer to the outer level of lexical nesting.

Figure 1 shows a possible layout for stack frame:



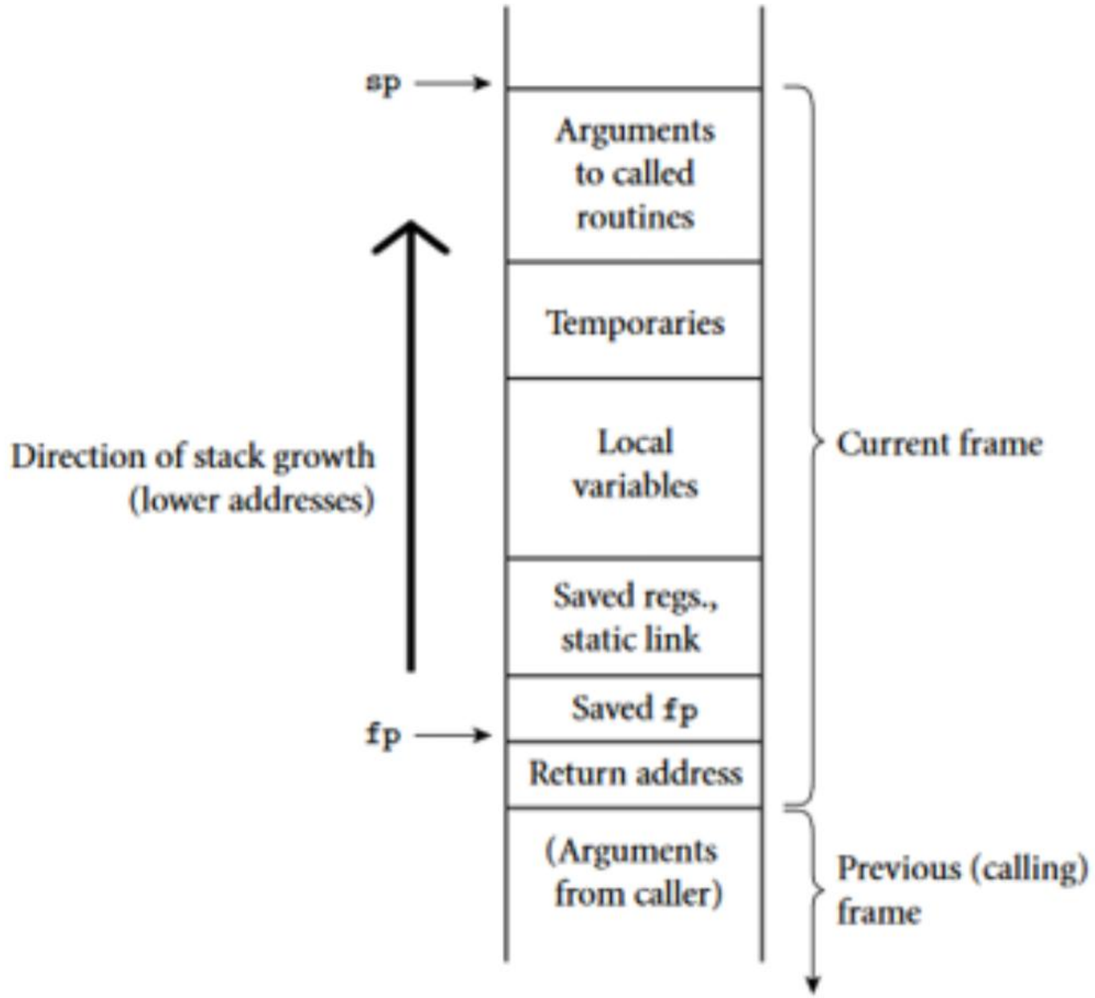


Figure 1: A Typical Stack Frame

The stack pointer (sp) is used to point to the first unused location on the stack. The frame pointer (fp) points to a location near the bottom of the frame. The space for all the arguments is reserved in the stack even if they are passed by the compilers in the registers.

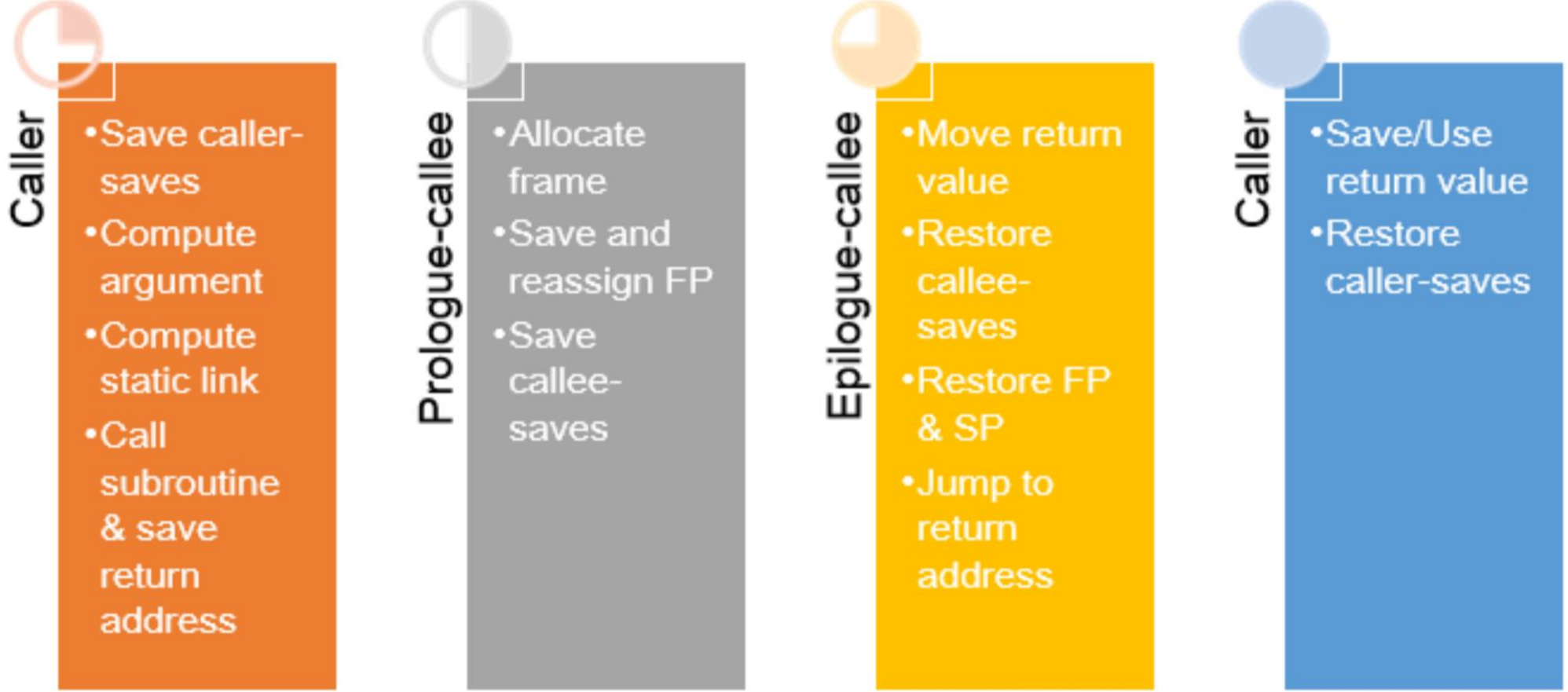
In order to maintain the stack layout, the calling sequence is operated in the following manner:

1. The caller saves any registers whose values will be required after the call.
2. The caller computes the values of arguments and moves them into the stack or registers.
3. The caller computes the static link and passes it as a hidden argument.
4. The caller uses a special subroutine call instruction to jump to the subroutine.

In its prologue, the procedure is as follows:

1. The callee allocates a frame by subtracting an appropriate constant from the sp.
2. The callee saves the old frame pointer into the stack and assigns it an appropriate new value.
3. The callee saves any callee-saves registers that may be overwritten by the current routine.

After the subroutine has completed, the epilogue moves the return value into a register or a reserved location in the stack. It restores callee-saves registers, if required, and restores the fp and the sp. It also jumps back to the return address. Finally, the caller moves the return value to wherever it is needed and restores caller-saves registers if needed.



4.3.1 In-line Expansion

Instead of stack-based calling conventions, various language implementations use subroutines that are expanded in-line at the point of call. A copy of the "called" routine becomes a part of the "caller" and no actual subroutine call occurs. The in-line expansion is used to avoid overheads, such as space allocation, branch delays from the call and return, maintain the

4.3.1 In-line Expansion

Instead of stack-based calling conventions, various language implementations use subroutines that are expanded in-line at the point of call. A copy of the “called” routine becomes a part of the “caller” and no actual subroutine call occurs. The in-line expansion is used to avoid overheads, such as space allocation, branch delays from the call and return, maintain the static chain or display. It is also used to perform code improvements, such as global register allocation, instruction scheduling, etc.

In some implementations, the compiler selects the subroutines to expand and which to compile. In some languages, the programmer can also make some specific routines to be in lined. In C++ and C99, the keyword inline is prefixed to a function declaration in the following manner:

```
inline int min(int x, int y) {return x>y ? x:y;}
```

There is a similarity between the in-line expansion and macros. However, the in-line expansion is more neutral, which means that it is an implementation technique that has no effect on the meaning of the program. The in-line expansion is responsible for increasing the size of the code which becomes a disadvantage. This expansion cannot be used for recursive subroutines.

4.3.2 Parameter Passing

Most of the subroutines are parameterised, which means they take arguments that are used to control the certain aspects of their behaviour. Parameter names that are displayed in the declaration of a subroutine are known as formal parameters. Parameters that are passed to a subroutine in a particular call are known as actual parameters.

Example 1: Formal and actual parameters

```
1. int calc(int a, int b){        //Function Definition

2.   return (a+b);}
```

```
3. int main(){
4.  calc(4, 5);          //Function call
5.  return 0;}
```

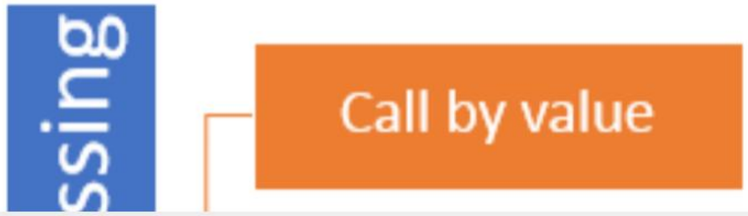
The variables a and b declared in the argument list of the function at line 1 are called as the formal parameters, whereas the values 4 and 5 passed to the function at line 4 are called as actual parameters, i.e. formal parameters are passed in function/method definition and when the method/function is called, actual parameters are passed.

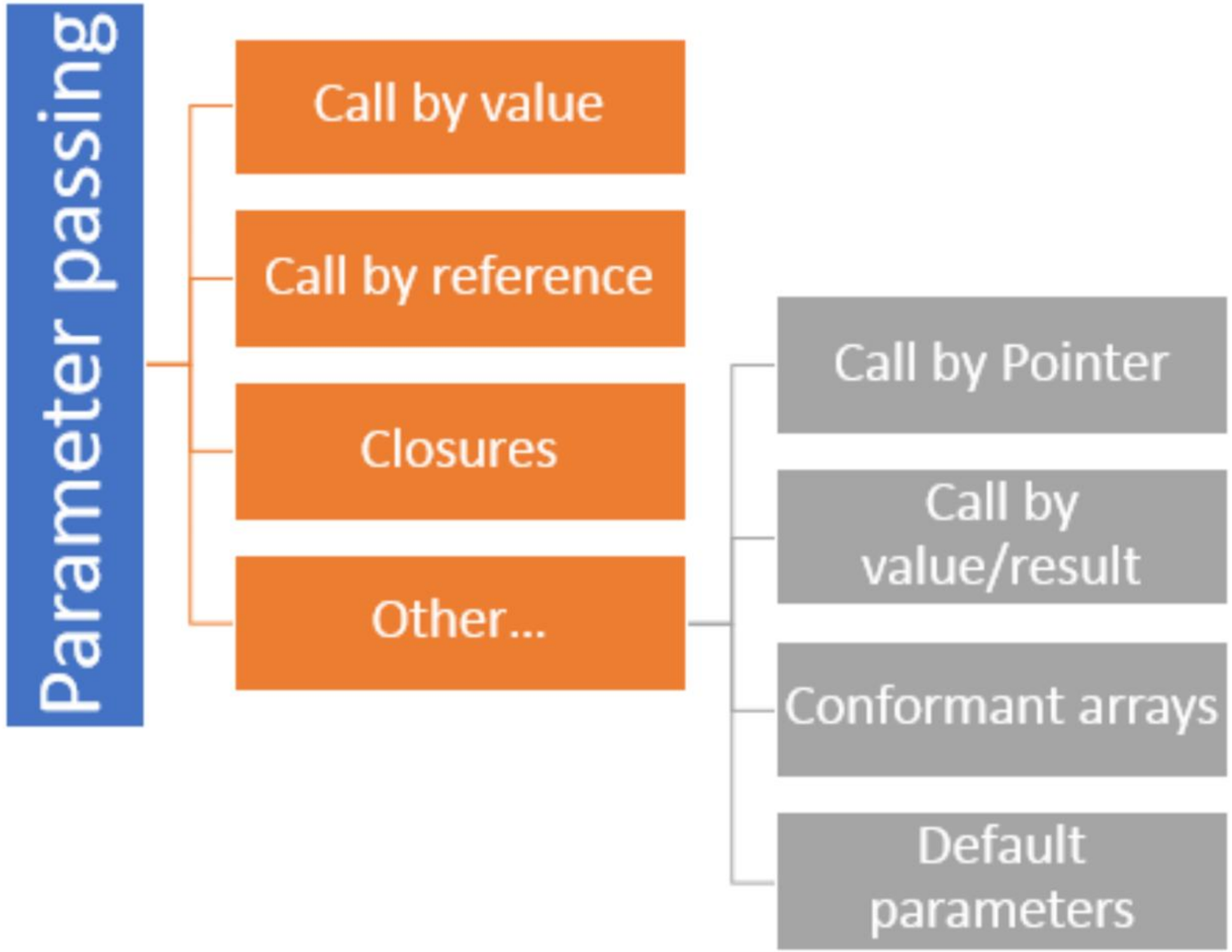
If arguments are to be used by a function, the variables accepting the values of the arguments must be declared by the function. These variables are called formal parameters of the function and are written in function header. They are local in nature, i.e. they become active when the function is called and inactive when call is over.

On the other hand, the parameters provided with function call are actual parameters. These parameters are written under parenthesis when a function is called. Example 1 shows the call or invoke the method in C++:

Earlier actual parameters were referred to as arguments. In the following sections, you will learn about the common parameter-passing modes that are implemented by passing values, references or closures.

Some languages use a prefix notation for calls to user-defined subroutines where the subroutine name is followed by a parenthesised argument list. For example, in Lisp, the function name is specified inside the parenthesis, such as (min x y).




Parameter passing
Call by value
Call by reference
Closures
Other…
Call by Pointer
Call by value/result
Conformant arrays
Default parameters

4.3.3 Call by Value

In this method, we are passing arguments, i.e. the actual parameters to the function. These actual parameters are stored into the formal parameters of the function and the changes made to the formal parameters do not affect the value of actual parameters, which means the original values remain unchanged.

Call by value is the default method for passing values during the calling of a function. This method passes the copies of the actual values of the arguments into the formal arguments of the function definition. This method does not change the value of the argument within the function. In other words, if the value of the formal arguments changes in the called function then it does not have any effect on the values of the actual arguments in the calling function. Here, actual values are safe. This method is also known as pass by values. The simple statement like “sum (a, b)” is an example of the call by value or pass by value in the calling function.

Example 2 demonstrates how to call function by value in C++:

Example 2: Call Function by Value

```
#include<iostream>

int add_func(int no);

using namespace std;

int main()

{
```

```
        int num,res;
        num=5;
        cout << "The initial value of the number: " << num << endl;
        res=add_func(num);
        cout << " The final value of the number: " << num << endl;
        cout << " The total value is: " << res << endl;
        return(0);
    }
    int add_func(int num)
    {
        num=num+150;
        return(num);
```

```
}
```

Figure 2 displays the output of Example 2:

```
The initial value of the number: 5
 The final value of the number: 5
 The total value is: 155
```

Figure 2: Output of Example 2

In Example 2, the value of the variable before calling the function is 5. When you made the function call, it adds 150 to the parameter number. When the function is returned, the result will have the added value. The value of the variable num before and after calling the function remains the same as 5. This indicates that changes made to the parameter inside the add_func function do not affect the arguments passed to it.

Example 3 defines swapping of numbers for explaining call by value in C language. The function name swap() takes two arguments and does not return any value. In the main() function, the scanf() function reads two values and printf() function prints this. Now, the swap() function is called and these two values are passed to the function. After that, the printf() function again prints it.

Example 3: Call by Value in C

```
/* Call by value */
```

```
#include<stdio.h>
#include<conio.h>
void swap(int x, int y)
{
      int t;
      t = x;
      x = y;
      y = t;
      printf("\n Within function after swapping x = %d \t y =%d
",x,y);
}
int main()
{
      int x, y;
      printf("Enter two values : ");
```

```
    scanf("%d %d", &x, &y);
    printf("\n Before calling x = %d \t y = %d",x,y);
    printf("\n \n Call by value\n ********\n");
    swap(x,y);
    printf("\n After calling x = %d \t y = %d ",x,y);
    return 0;
}
```

Figure 3 displays the output of Example 3:

```
Enter two values : 12 15

 Before calling x = 12   y = 15

 Call by value
 ********
```

```
Call by value
********

Within function after swapping x = 15   y =12
After calling x = 12    y = 15
```

Figure 3: Output of Example 3

4.3.4 Call by Reference

In this method, we are passing reference of the actual parameters to the function, which stores the reference of actual parameters into the formal parameters. It uses the reference to access actual parameters inside the function. The changes made to the formal parameters affect the actual parameters. So, if you want to change the value of actual parameters during function call, use call by reference. The changes in the value of actual parameters remain intact.

Call by reference is the second method for passing values during calling a function that copies the address of the actual argument into the formal arguments. This method passes the address of the actual values of arguments into the formal arguments of the function definition or function call. This method can change the value of the argument within the function. In other words, if values of the formal arguments change in the called function then it effects the values of the actual arguments in the calling function.

Here, actual values are not safe and by accessing the addresses of the actual arguments, it can be changed from the called function. This method is also known as the pass by references. The simple statement, such as “sum(&a, &b)”, is an example of call by reference or address in the calling function where the operator ‘&’ is used. For accessing the value at this address, the dereferencing operator or pointer [*] is used in the function definition.

Example 4 demonstrates how to swap to number by calling the function by reference in C++ language:

Example 4 demonstrates how to swap to number by calling the function by reference in C++ language:

Example 4: Call by Reference in C++

```
#include<iostream>

#include<conio.h>

// Function prototype

using namespace std;

void swap(int&, int&);

int main()

{

      int x = 5, y = 10;

      cout << "Before swapping" << endl;

      cout << "x = " << x << endl;

      cout << "y = " << y << endl;

      swap(x, y);

      cout << "\nAfter swapping" << endl;
```

```
        cout << "\nAfter swapping" << endl;
        cout << "x = " << x << endl;
        cout << "y = " << y << endl;
        getch();
        return 0;
}
void swap(int& num1, int& num2) {
        int temp;
        temp = num1;
        num1 = num2;
        num2 = temp;
}
```

Figure 4 displays the output Example 4:

```
Before swapping
```

```
Before swapping
x = 5
y = 10

After swapping
x = 10
y = 5
```

Figure 4: Output of Example 4

Example 5 defines the same example of swapping of numbers for explaining call by address. The function name is swap(). It takes two arguments and does not return any value. In the main() function, the scanf() function reads two values and the printf() function prints this. Now, the swap() function is called and the address of two values are passed to the function. After that, the printf() function again prints the numbers.

Example 5: Call by Reference in C

```
/* Call by reference / address */

#include<stdio.h>

#include<conio.h>
```

```
void swap(int *x, int *y)
{
      int *t;
      *t = *x;
      *x = *y;
      *y = *t;
      printf("\n After swapping within function x = %d \t y = %d
",*x,*y);
}
int main()
{
      int x, y;
      printf("Enter two values : ");
```

```
    scanf("%d %d", &x, &y);
    printf("\n Before calling function x = %d \t y = %d",x,y);
    printf("\n \n Call by reference/address\n ********\n");
    swap(&x,&y);
    printf("\n After calling function x = %d \t y = %d",x,y);
    return 0;
}
```

Figure 5 displays the output of Example 5 where two numbers 12 and 15 are entered and printed. After calling the function, the `printf()` function prints the same values 12 and 15 for x and y, respectively. Meanwhile, the `swap()` function also changes the value of x that affects the actual value of variables x and y. The line "After swapping within function x = 15 y = 12" is showing the swapped values and the line "After calling function x = 15, y = 12" is showing the actual values, which proves that call by reference affects the actual values of the function call.

```
Enter two values : 12 15

 Before calling function x = 12          y = 15
```

```
Enter two values : 12 15

 Before calling function x = 12          y = 15

 Call by reference/address
 ********

 After swapping within function x = 15   y = 12
 After calling function x = 15   y = 12
```

Figure 5: Output of Example 5

Call by Value	**Call by Reference**
Variable value is passed	Variable reference is passed
The formal parameters create a separate copy of the actual variables	The formal parameters introduce a name for the actual variables
Any modifications performed on the formal	Modifications performed using the

Call by Value	Call by Reference
Variable value is passed	Variable reference is passed
The formal parameters create a separate copy of the actual variables	The formal parameters introduce a name for the actual variables
Any modifications performed on the formal parameters do not affect the actual parameters	Modifications performed using the reference affect the actual parameters
Address or pointer is not required	Address or pointer is required

4.4 Closures

A closure — a reference to a subroutine, together with its referencing environment — may be passed as a parameter for any of several reasons. The most obvious of these arises when the parameter is declared to be a subroutine, sometimes called a formal subroutine.

Example xx: A java script closure and its usage

```
function getMultiplier(multiplyBy) {
    return function(num) {
        return multiplyBy * num;
    }
}
var multiplyByTwo = getMultiplier(2);
var multiplyByTen = getMultiplier(10);
var twoIntoFive = multiplyByTwo(5);
var tenIntoSix = multiplyByTen(6);
console.log(twoIntoFive);   // prints 10
console.log(tenIntoSix);    // prints 60
```

4.4.1 Other

These include many variations of the call by reference and call by value passing modes as well as a few language specific modes. Some of them are listed below:

4.4.2 Call by Pointer

In this method, you copy the address of actual parameters into the formal parameters. It uses the address to access the actual parameters used to call the function. In call by pointer, the pointers of actual parameters are passed to the called function. Therefore, function parameters should also be declared as pointer types. This method is pretty similar to call by reference. The changes made to the formal parameters affect the actual parameters and the changes in the value of actual parameters remain intact.

Example 6 demonstrates how to swap two numbers by calling the function by pointer:

Example 6: Call by Pointer in C++

```
#include<iostream>

// Function prototype

void swap(int *, int *);

using namespace std;

int main()
```

```
int main()
{
    int x = 5, y = 10;
    cout << "Before swapping using pointer" << endl;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    swap(&x, &y);
    cout << "\nAfter swapping using pointer" << endl;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    return 0;
```

```
}

void swap(int *num1, int *num2) {
    int temp;
    temp = *num1;
    *num1 = *num2;
    *num2 = temp;
}
```

Figure 6 displays the output Example 6:

```
Before swapping using pointer
x = 5
y = 10
```

```
After swapping using pointer
x = 10
y = 5
```

Figure 6: Output of Example 6

4.4.3 Call by Value/Result

Call-by-reference allows the called routine to modify the actual parameter. This can also be achieved by using call-by-value/result, a mode first introduced in Algol W. Like call-by-value, call-by-value/result copies the actual parameter into the formal parameter at the beginning of subroutine execution. But unlike call-by-value, it also copies the formal parameter back into the actual parameter when the subroutine returns.

Example xx: Call by Value/result

```
1.    x : integer                    //global
2.    procedure foo (y : integer)
3.    y := 3
4.    print x
5.
```

```
5.
6.      x := 2
7.      foo(x)
8.      print x
```

Output:

```
2
3
```

Explanation:

In the above example, value/result would copy x into y at the beginning of foo, and y into x at the end of foo. Because foo accesses x directly in-between, the program's visible behaviour would be different than it was with call-by-reference or call-by-value. The difference can be examined by looking at line 6 where x is assigned the value 2. The function call at line 7 passes this value of x as 2 to the function foo where it is captured in variable y. The assignment of 3 into y would not affect x until after the inner print statement, i.e. at line 4 the value of x and y is different. After execution of line 4, the new variable value, i.e. value of y which is 3 is passed back to the original variable x in the program line, i.e. at the start of line 8, the value of x is the updated value as 3. So, the program would print 2 and then 3.

If the program were resolved using call-by-value, it would result into 2 and 2, whereas if the syntax supported call-by-reference, the output would be 3 and 3.

4.4.4 Conformant Arrays

A formal array parameter whose shape is finalised at run time in a language that usually determines shape at compile time is called a conformant or open, array parameter.

Explanation:

In the above example, value/result would copy x into y at the beginning of foo, and y into x at the end of foo. Because foo accesses x directly in-between, the program's visible behaviour would be different than it was with call-by-reference or call-by-value. The difference can be examined by looking at line 6 where x is assigned the value 2. The function call at line 7 passes this value of x as 2 to the function foo where it is captured in variable y. The assignment of 3 into y would not affect x until after the inner print statement, i.e. at line 4 the value of x and y is different. After execution of line 4, the new variable value, i.e. value of y which is 3 is passed back to the original variable x in the program line, i.e. at the start of line 8, the value of x is the updated value as 3. So, the program would print 2 and then 3.

If the program were resolved using call-by-value, it would result into 2 and 2, whereas if the syntax supported call-by-reference, the output would be 3 and 3.

4.4.4 Conformant Arrays

A formal array parameter whose shape is finalised at run time in a language that usually determines shape at compile time is called a conformant or open, array parameter.

4.4.5 Default parameters

A default parameter is one that need not necessarily be provided by the caller because if it is missing then a preestablished default value will be used instead.

4.5 Generic Subroutines and Modules

Subroutines are used to provide a way to perform operations on various object (parameter) values. In some programs, it also helps in performing an operation on various object types. An operating system uses queues to hold the processes, memory descriptors, file buffers, device control blocks, and other objects. In some languages, the standard methods to declare queue and deque subroutines need the items to be declared more statically. In languages, such as Pascal or Fortran, this method means that the programmer should create separate copies of enqueue and deque for each type. In languages, such as C, it is possible to define a queue of pointers to arbitrary objects.

Implicit parameter polymorphism is used to declare the subroutines who have parameter types unspecified but still type-safe. This approach has some drawbacks as in Lisp, it delays type checking until runtime. Another method is used to provide an explicitly polymorphic generic method that allows creating a collection of similar subroutines with different types from a single copy of the source code. Languages that use generics are Ada, C++, Clu, Eiffel, Modula-3, Java and C#.

Generic modules and classes are used to create containers that are data abstractions to hold a collection of objects. Example of such containers are stack, queue, heap, set and dictionary abstractions implemented as lists, arrays, trees or hash tables. A generic subroutine is needed in generic modules. The generic parameter depends on the languages. Java and C# can only pass types while Ada and C++ are more general. They allow values that are ordinary types, including subroutines and classes.

4.5.1 Implementation

A template in C++ is similar to a blueprint that creates a generic class or a function that reduces code duplication and includes independent writing of a code of any particular type. Templates are the base of generic programming, which involves code reuse and enables a programmer to write a code that behaves in the same way for any type of data. Templated class does not depend on the data type it deals with. Syntax to declare template is as follows:

template<parameter list> declaration

Here, template is a keyword, parameter list is a list of template parameters and declaration is declaration of class or function. The following syntax specifies how to declare a templated class in C++:

template<parameter list> declaration

Here, template is a keyword, parameter list is a list of template parameters and declaration is declaration of class or function. The following syntax specifies how to declare a templated class in C++:

template <class my_type> class my_class {...};

Here, my_type is an identifier that will be specified when the object of the class is created. This identifier could be used to define the variables in the class. For example:

my_type my_variable

4.5.2 Generic Functions

The special functions that specify a group of functions which can work with generic types are referred to as function templates. You may not have to repeat the whole code for every type if you use a function template as it can be altered to one or more types or classes. The syntax for function template is as follows:

template <class type> return-type function-name(parameter list) {

// code here

}

In the above syntax, type is a placeholder for a data type which can be used within the function definition.

Example 7 demonstrates the function template:

Example 7: Generic Function

```
#include <iostream>

#include<conio.h>

#include<string.h>

using namespace std;

template <class T>

inline T const & Max (T const& a, T const& b) {

      return a < b ? b:a;

}

int main () {

      int m = 40;

      int n = 35;

      cout << "Max(m, n): " << Max(m, n) << endl;
```

```
    double d1 = 13.5;

    double d2 = 20.7;

    cout << "Max(d1, d2): " << Max(d1, d2) << endl;

    getch();

    return 0;

}
```

Figure 7 displays the output of Example 7:

```
Max(m, n): 40
Max(d1, d2): 20.7
```

Figure 7: Output of Example 7

4.5.3 Generic Class

We can define a class template in the same way as the function template. A class template contains members that use template parameters as types and are useful for defining container classes. The syntax for class template is as follows:

template <class type> class class-name{

// code here

}

In the above syntax, type is a placeholder for a data type which can be used when you instantiate the class and define one or more generic data type by using comma separated list.

Example 8 demonstrates the use of the class template:

Example 8: Generics Class

```
#include <iostream>

#include<conio.h>

using namespace std;

template <class T>

class myval {

      T a, b;
```

```
        public:
        myval (T first, T second)
        {a=first; b=second;}
        T getmaxval ();
    };
    template <class T>
    T myval<T>::getmaxval ()
    {
        T returnval;
        returnval = a>b? a : b;
        return returnval;
```

```
}
int main () {
      myval <int> myobject (155, 145);
      cout << "The maximum value is: " <<myobject.getmaxval()<<
endl;
      getch();
      return 0;
}
```

Figure 8 displays the output of Example 8:

```
The maximum value is: 155
```

Figure 8: Output of Example 8

4.5.4 Template Parameters

A template is an entity defining a group of classes, group of functions or group of variables. One or more template parameters can be passed in a parameter list as shown in the syntax of template given below:

template < parameter-list > declaration

Templates can have the following three types of parameters:

- Type Template Parameters
- Non-type Template Parameters
- Template Template Parameters

4.5.5 Type Template Parameters

Type template parameters are the most common type of template parameters. The syntax for a type template parameter is as follows:

typename name

or

class name

where name (optional) is a valid symbol name which can be introduced in the template parameter list. You can refer to this name within the code as a generic type and declare variables, arrays or pointers. The actual data type for this name is specified during object creation.

For instance:

```
template <typename T>
```

```
template <typename T>
void myfunction(T value)
{
      const T& ref=value;
      T* ptr=new T;
      T temp(10);
}
```

Here, name of the template is T and void myfunction is a function that accepts the value of type T. The code for the function is enclosed within { }.

4.5.6 Non-Type Template Parameters

Non-type template parameters include values rather than the types. They can be of any value, such as a compile-time constant. The non-type template parameters can include built-in types along with generic data types. The syntax for a non-type template parameter is as follows:

```
template<int m>
```

```
template<int m>

class x{};

template<double* myval>

class y{};

template<void (*func)(int)>

void z()

{}
```

In the above syntax, the class template x has a non-type template parameter of integer, class template y has a non-type template parameter of pointer-to-double and function template z has a non-type template parameter that returns void by referring a pointer to a function, with a single integer parameter.

For instance:

```
template <class T, int size>

class arr

{
```

```
        T array[size];

    }

    arr <int, 5> array1; // Array of 5 integers

    arr<float, 10> array2; // Array of 10 floats

    arr<char, 50> array3; // Character string of size 50
```

The size of the array is supplied by this template as a parameter. At the time of compilation, the size of the array is known to the compiler.

4.5.7 Template Template Parameters

Template template parameters determine a template to be parameterised by another template name. The name of the class template is the template parameter for a template template parameter.

Only keyword class is used while declaring the template template parameter, while both class and typename can be used in type template parameter declaration. For example:

```
template<template <class T> class X> class A { };

template<class T> class B { };
```

For instance:

For instance:

```
template<template<typename T> class ContainerType>

class ClassDemo

{

      ContainerType<int> intContainer;

      ContainerType<std::string> stringContainer;

      // code here

};
```

4.6 Exception Handling

An exception is a situation where a program produces an unexpected error during the execution of the program. It is important to have a mechanism that can deal with these exceptions. Exception handling is a technique that separates error handling code from the main code of the program and thereby makes it easier for the user to read and write the code. It transfers the control from one point of a program to another. An exception can occur because of any of the following conditions:

- An attempt to divide by zero
- Accessing an array outside of bounds
- Running out of memory
- Running out of disk space

Some of the types of exceptions are:

1. **Checked Exception**: This is also known as compile time exception because it occurs at compile time. For example, io exception, SQL Exception, etc.
2. **Unchecked Exception**: It is also known as run-time exception because it occurs at run time. For example, arithmetic exception, number format exception, etc.
3. **Error**: Errors are not possible to handle in code. If an error is in the code, error message will generate at compile. For stack overflow, etc.

The three keywords to handle exceptions in C++ and Java are as follows:

- **Try**: It is a block of program code where an exception is detected. As soon as the exception is detected, try transferring the program control to the catch block by throwing the exception.
- **Catch**: It determines the type of exception and provides a block of code which will execute when an exception is thrown. There can be multiple catch blocks in a program depending upon the type of exception raised. Catch is responsible for taking corrective actions for exceptions.
- **Throw**: It is used to throw an exception when a problem occurs and can be called from anywhere in the program. The point where throw is executed is referred to as throw point.

4.6.1 Try and Catch Keywords

Try and Catch are the two keywords which are related to each other and are most useful in exception handling. The first keyword is try. When we determine the block of code that can cause exception, we write this block of code within try and is called the try block. In Java, the try block is followed by either the catch block or finally block. The syntax is:

```
try {
      // codes that may raise exception
} catch (ExceptionTypeOne exObj) {}
OR
try {
      // codes that may raise exception
}
finally () {}
```

The second keyword is catch. When an exception occurs in the try block, the flow of control jumps into the catch block. In the catch block, we write the code for handling the exception, i.e. what the program should do if exception occurs. The syntax in Java is:

catch (ExceptionTypeOne exObj) {

// Codes of exception handler for ExceptionTypeOne

}

In C++, the role of a catch block is to catch the exception thrown by the try block. Therefore, it must follow immediately after the try block that throws the exception. The declaration of exception is enclosed within the round brackets as shown in the syntax given below:

```
try
{
      // protected code
}
Catch (ExceptionName excp )
{
      //Code for handling ExceptionName exception
```

```
}
```

Here, ExceptionName is a type of exception and the catch catches the exception of ExceptionName type.

If you want to declare a catch block that can catch any type of exceptions thrown by the try block, you need to keep three dots (ellipsis) within round bracket following the catch block as shown below:

```
try
{
      // protected code
}
Catch ( . . . )
{
      // handle an exception here
}
```

Let us now understand the usage of try and catch keywords with the help of a Java program. In Example 10, a Java program is created to handle an arithmetic exception. In this

Let us now understand the usage of try and catch keywords with the help of a Java program. In Example 10, a Java program is created to handle an arithmetic exception. In this program, we can see that if an exception occurs in the try block, the flow of program jumps to the catch block. The remaining code in try block is not executed. If the exception object sent by the try block matches with the object in the catch block, then the catch block is executed and the exception is handled.

Example 9: Exception Handling Example

```
class ExampleExceptionHandling{
    public static void main(String args[]){
        try{
            int i = 12/0; // this line can raise exception
            System.out.println("Codes after exception in try
        block");
        }
        catch(ArithmeticException obj){
            System.out.println(obj);
        }
```

```
            System.out.println("Codes after catch block");

        }

    }
```

Figure 9 displays the output of Example 9:

```
E:\Core Java\programs\exception>javac ExampleExeptionHandling.java

E:\Core Java\programs\exception>java ExampleExeptionHandling
java.lang.ArithmeticException: / by zero
Codes after catch block

E:\Core Java\programs\exception>
```

Figure 9: Output of Example 9

Example 10 demonstrates how to catch exceptions in C++:

Example 10: Catching Exceptions in C++:

```
#include <iostream>

using namespace std;
```

```
int main()
{
    int x,y,z;
    float p;
    cout << "Enter the value of x:";
    cin>>x;
    cout << "Enter the value of y:";
    cin>>y;
    cout << "Enter the value of z:";
    cin>>z;
    try
    {
```

```
try
{
      if((x-y)!=0)
      {
            p=z/(x-y);
            cout<<"Result is:"<<p;
      }

    else
    {
          throw(x-y);
    }
```

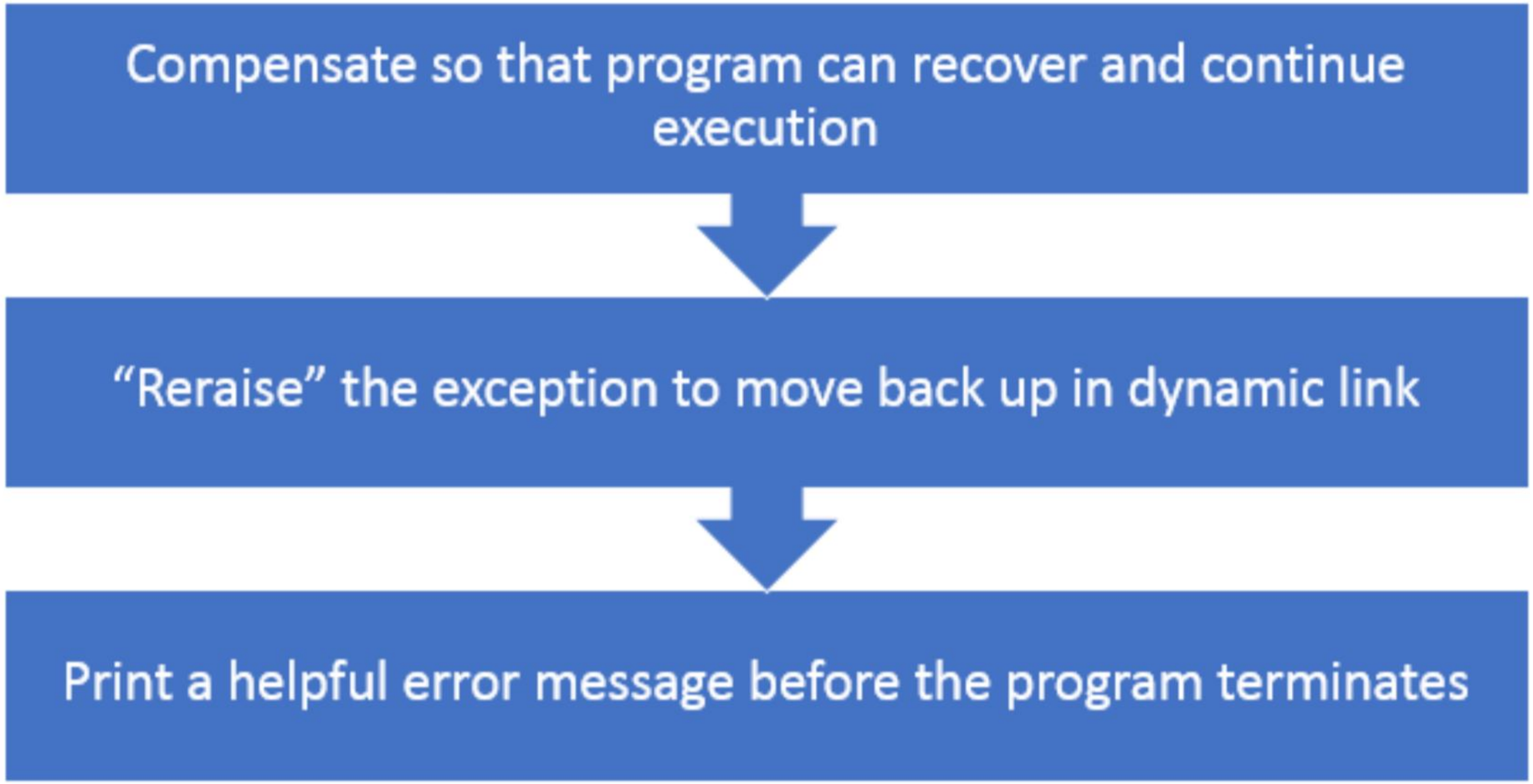
```
        }
        catch (int u)
        {
            cout<<" Result is infinite because x-y is:"<<u;
        }
}
```

When you run Example 10, it will produce the following output:

```
Enter the value of x:10
Enter the value of y:10
Enter the value of z:20
 Result is infinite because x-y is:0
```

Figure 10: Output of Example 10

So, in general, handler performs one of the 3 types of operations



Print a helpful error message before the program terminates

4.6.2 Multiple Catch Block

Java provides us this facility to write multiple catch block for single try block. If we write multiple instructions in try block then there is a chance to occur multiple types of exception. For handling these exceptions, we need to write multiple catch blocks. In Example 11, we illustrate how to write multiple catch blocks for single try block:

Example 11: Multiple Catch Block

```
class ExampleMultiCatch{
    public static void main(String args[]){
        try{
            int i = 12/0; // this line can raise exception
            System.out.println("Codes after exception in try
block");
```

```
            }
            catch(ArithmeticException obj){

                System.out.println("perform first job");
            }
            catch(Exception obj){

                System.out.println("perform second job");
            }
            System.out.println("Codes after catch block");
        }
    }
```

When you run Example 11, it will produce the following output:

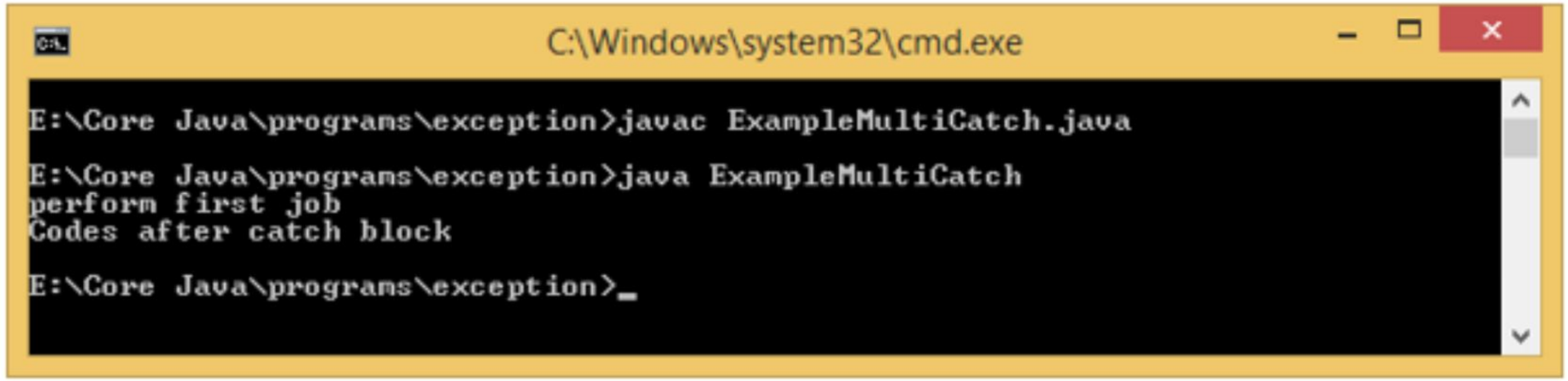


Figure 11: Output of Example 11

A nested try block means a try block within a try block. In the nested try block when an exception occurs at outer try block, the flow of program jumps to outer catch block, the inner try and catch block is not executed. For example, in Example 12, nested try block is used and the exception occur in outer try block.

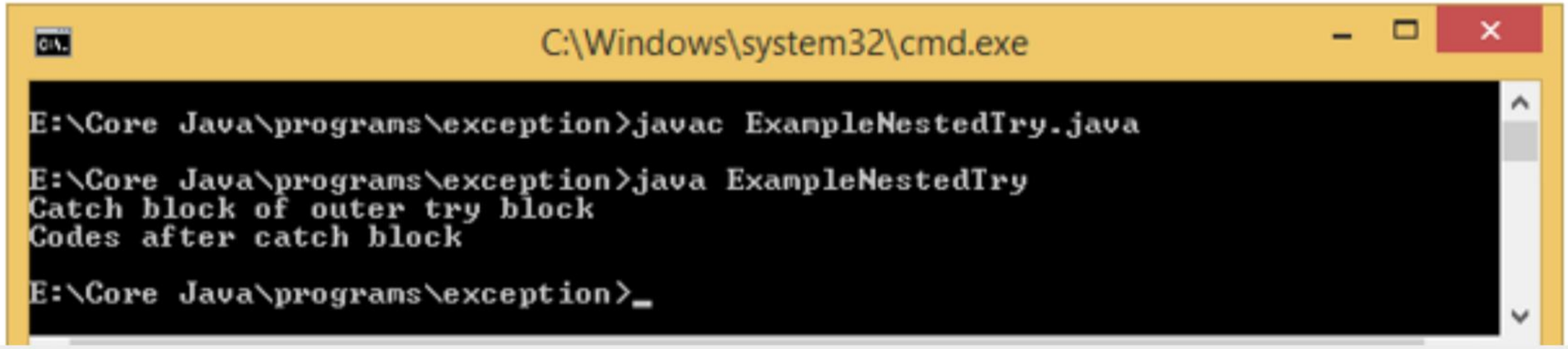
Example 12: Nested Try Block

```
class ExampleNestedTry{
```

```
class ExampleNestedTry{
      public static void main(String args[]){
            try{
                  int j = 12/0;
                  try{
                        int i[] = {9};
                        i[19] = 90;// this line can raise exception
                        System.out.println("Codes  after  exception  in
try block");
                  }
                  catch(ArrayIndexOutOfBoundsException obj1){
                        System.out.println("Catch  block  of  inner  try
block");
                  }
            }
```

```
catch(ArithmeticException obj2){
System.out.println("Catch block of outer try
block");
}

System.out.println("Codes after catch block");
}
}
```

When you run Example 12, it will produce the following output:



4.6.3 Throwing an Exception

When an exception is raised in a program, it can be thrown to an exception handler by using the throw statement. This statement determines the type of the exception and throws it to the appropriate catch block that can handle it. The throw statement should be declared inside the try block, not outside it.

For example, in C++:

```
double divide(double x, double y)
{
    if (y == 0) //it is division by zero. . .
        throw "This is a division by zero exception!!!";
    else
        return x / y;
}
```

In the above code, the divide() function throws an exception if the denominator, y, is zero. The exception will be thrown when this function is called with value of y as 0 as shown below:

```
try
```

```
{
        double res = divide(2, 0);
}
        catch (char* z)
{
        cout <<"The error is: "<< z<<endl;
}
```

The code produces the following output:

The error is: This is a division by zero exception!!!

The throw keyword in Java is used to throw an exception explicitly. With the help of this keyword, you can show custom message when an exception occurs. The throw keyword is mainly used with a user-defined exception. It is used to declare an exception. The throw keyword is also used before a method which can raise an exception.

4.7 Coroutines and Events

After understanding the layout of the run-time stack, it's time to consider the implementation of general control abstractions, such as coroutines. A coroutine is represented by a closure where one can jump using the nonlocal go to, which is known as transfer. The main difference between the two abstractions is that continuation is a constant, which means it does not change when it is created once. When you move to a continuation, the old program counter is lost until a new continuation is created to hold it. When you transfer from one coroutine to another, the old program counter is saved. Therefore, it is possible to go into the same continuation multiple times where each jump will start at precisely the same location.

Coroutines are the execution contexts that exist together but execute one at a time. They transfer their control to each other by name. Coroutines are used to implement iterators and thread. Threads appear in languages, such as Algol 68, Modula (1), Modula-3, Ada, SR, Occam, Java, and C#. Here is a simple example of an application where coroutines might be useful. Suppose you are creating a screensaver program that uses a black picture for an inactive workstation and it also keeps the picture moving to avoid phosphor or liquid-crystal "burn-in". Now, you want your screen saver to perform the "sanity checks" on the file system in the background to check for corrupted files. The problem with this method is that successive sanity checks are dependent on each other. Another approach is to cast the operations as coroutines.

4.7.1 Stack Allocation

The coroutines are concurrent therefore they cannot share a single stack. It means that subroutine calls and returns which do not occur in last-in-first-out order. If a coroutine is declared at the outermost level of lexical nesting then the stacks are entirely disjoint. It means they only share the objects that are global and statically allocated. It is not easy to allocate an arbitrary number of stacks. Therefore, an easy solution is to give each coroutine a fixed amount of statically allocated stack space. This approach is used in Modula-2, where a programmer can specify the size and location of the stack when initialising a coroutine.

Stack frame allocation from heap in Lisp and Scheme can help in avoiding the problems of overflow and internal fragmentation. It also increases the overhead on each subroutine call. Another method is to allocate the stack in large fixed-size "chunks". At each call, the calling sequence checks that if there is sufficient space in the current chunk to hold the frame. If there is not enough space, another chunk is allocated for the frame. Every time the subroutine returns, the epilogue code checks if the current one is the last frame, then the chunk is returned to a "free chunk" pool.

returned to a “free chunk” pool.

4.7.2 Transfer

To transfer from one coroutine to another, the run-time system should change the program counter (PC), the stack and the contents of the processor’s registers. These changes are included into the transfer operation, where when one coroutine call transfers, a different one is returned. Since the changes are happening inside the transfer, changing the PC from one coroutine to another needs the right return address. If the transfer saves the return address in the stack then the PC will change automatically.

The stacks can be changed by simply changing the stack pointer register and to avoid using the frame pointer inside of transfer itself. At the beginning of a transfer, the return address is pushed along with the callee-saves registers onto the current stack. Then we change the sp, pop the (new) return address (ra) and other registers off the new stack.

4.7.3 Events

Event occurs only when the user tries to bring about a change in the state of the object. The process of organising these events in a proper sequence is known as event handling. Event occurs when the user tries to communicate with the Graphical User Interface (GUI). Event handling helps in executing the events in a proper order. Clicking the mouse, pressing the key to the keyboard, releasing the key of the keyboard, etc., are all examples of event handling.

In the I/O operations, a program looking for input will require it explicitly and will wait if it is not available. The synchronous and blocking input is not acceptable in modern applications with graphical interfaces. Instead, a handler is needed when a given event occurs. They are known as call back functions because the run-time system calls back into the main program.

4.7.4 Sequential Handlers

Event handlers in sequential programming languages are implemented as the “spontaneous” subroutine calls using a mechanism defined and implemented by the operating system. In order to receive events using this mechanism, a program invokes a setup handler library routine and passes as argument the subroutine it wants to have invoked when the event occurs

4.7.4 Sequential Handlers

Event handlers in sequential programming languages are implemented as the “spontaneous” subroutine calls using a mechanism defined and implemented by the operating system. In order to receive events using this mechanism, a program invokes a setup handler library routine and passes as argument the subroutine it wants to have invoked when the event occurs.

In hardware, any asynchronous device activity during the P’s execution triggers an interrupt mechanism that saves P’s registers, switches to a different stack and jumps to a predefined address in the OS kernel. Similarly, if another process Q is running when the interrupt occurs, the kernel saves the P’s state at the end of its last time slice.

4.7.5 Thread-based Handlers

In modern programming languages, events are handled by a separate thread of control instead of spontaneous subroutine calls. When there is a separate handler thread, the input can be synchronous which means the handler thread makes a system call to request the next event and waits for it to occur. In the meantime, the main program executes. If the program wants to handle the multiple events concurrently, it may create multiple handler threads. In order to protect the shared data structures, the main program and the handler thread will generally require a full-fledged synchronisation mechanism.

In Java Swing, an event handler in Java is an instance of a class that implements the ActionListener interface with a method named actionPerformed:

```
class PauseListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        // do needful
    }
```

```
class PauseListener implements ActionListener {
      public void actionPerformed(ActionEvent e) {
            // do needful
      }
}
...
JButton pauseButton = new JButton("pause");
pauseButton.addActionListener(new PauseListener());
```

4.8.1 Recap

Let us quickly recapitulate the following discussion points:

- A subroutine is a principal mechanism for control abstraction in most programming languages.
- To maintain the subroutine call stack, the calling sequence is used.
- The saving and restoring of registers are essential during subroutine and calls since they determine the state of the programming environment.
- Instead of stack-based calling conventions, various language implementations use subroutines that are expanded in-line at the point of call.
- Call by value is the default method for passing values during the calling of a function.
- Call by reference is the second method for passing values during calling a function that copies the address of the actual argument into the formal arguments.
- Subroutines are used to provide a way to perform operations on various object (parameter) values.
- Generic modules and classes are used to create containers that are data abstractions to hold a collection of objects.
- The special functions that specify a group of functions which can work with generic types are referred to as function templates.
- A template is an entity defining a group of classes, group of functions or group of variables.
- An exception is a situation where a program produces an unexpected error during the execution of the program.
- Exception handling is a technique that separates error handling code from the main code of the program and thereby makes it easier for the user to read and write the code.
- Coroutines are the execution contexts that exist together but execute one at a time.

4.8.2 Test Your Skills- Multiple Choice Questions

1. Which of the following keyword is not used in exception handling?

a. try

4.8.2 Test Your Skills- Multiple Choice Questions

1. Which of the following keyword is not used in exception handling?

a. try
b. catch
c. thrown
d. finally

2. Which of the following keyword is used in exception handling to throw an exception manually?

a. try
b. catch
c. throw
d. finally

3. In which of the following methods, the changes made to the formal parameters do not affect the value of actual parameters?

a. Call by value
b. Call by reference
c. Call by pointer
d. Call by subroutine

4. In which of the following methods, the reference of the actual parameters to the function stores the reference of actual parameters into the formal parameters?

a. Call by value
b. Call by reference

4. In which of the following methods, the reference of the actual parameters to the function stores the reference of actual parameters into the formal parameters?

a. Call by value
b. Call by reference
c. Call by pointer
d. Call by subroutine

5. Which of the following parameters can include built-in types along with generic data types?

a. Type Template Parameters
b. Non-type Template Parameters
c. Template Template Parameters
d. Module Template Parameters

4.8.3 Fill in the Blanks

1. _____ is the default method for passing values during the calling of a function.
2. _____ is the method for passing values during calling a function that copies the address of the actual argument into the formal arguments.
3. A _____ contains members that use template parameters as types and are useful for defining container classes.
4. A _____ is an entity defining a group of classes, group of functions or group of variables.
5. The _____ statement determines the type of the exception and throws it to the appropriate catch block that can handle it.

4.8.4 Theoretical Questions

c. Template Template Parameters
d. Module Template Parameters

4.8.3 Fill in the Blanks

1. _____ is the default method for passing values during the calling of a function.
2. _____ is the method for passing values during calling a function that copies the address of the actual argument into the formal arguments.
3. A _____ contains members that use template parameters as types and are useful for defining container classes.
4. A _____ is an entity defining a group of classes, group of functions or group of variables.
5. The _____ statement determines the type of the exception and throws it to the appropriate catch block that can handle it.

4.8.4 Theoretical Questions

1. What is the role of catch block in exception handling? How do you declare a catch block that can accept any kind of exceptions thrown by try block?
2. What is an exception? What is the use of exceptions handling?
3. What is a subroutine calling sequence?
4. What is the difference between formal and actual parameters?
5. What is the principal purpose of generics?

Chapter 5 : Data Abstraction in Object Orientation

Learning Objectives

- Explain the architecture of OOP
- Revise the features of object and class
- Understand abstraction and encapsulation and their relation
- Describe inheritance and its types
- Discuss polymorphism and its types
- Explain the use of virtual function
- Differentiate between overloading and overriding

5.1 Introduction

Object Oriented Programming (OOP) concept is a programming language model that is associated with classes and objects, which in turn revolve around the concepts of inheritance, polymorphism, abstraction and encapsulation. In this chapter, we will understand the implementation of these concepts in programs.

As you know that C++ language is based on classes and objects, which consist of various data members and member functions that operate with or within the two. In this chapter, we are going to discuss two member functions — constructors and destructors — which play a vital role in C++ programming. A constructor is a special member function of a class and is executed whenever new objects of that class are created. On the other hand, a destructor works opposite to a constructor. It is also a special member function and is executed every time an object of its class goes out of scope or the object is no longer required.In this chapter, you will also learn about the object-oriented programming. You will also learn about

- Explain the architecture of OOP
- Revise the features of object and class
- Understand abstraction and encapsulation and their relation
- Describe inheritance and its types
- Discuss polymorphism and its types
- Explain the use of virtual function
- Differentiate between overloading and overriding

5.1 Introduction

Object Oriented Programming (OOP) concept is a programming language model that is associated with classes and objects, which in turn revolve around the concepts of inheritance, polymorphism, abstraction and encapsulation. In this chapter, we will understand the implementation of these concepts in programs.

As you know that C++ language is based on classes and objects, which consist of various data members and member functions that operate with or within the two. In this chapter, we are going to discuss two member functions — constructors and destructors — which play a vital role in C++ programming. A constructor is a special member function of a class and is executed whenever new objects of that class are created. On the other hand, a destructor works opposite to a constructor. It is also a special member function and is executed every time an object of its class goes out of scope or the object is no longer required.In this chapter, you will also learn about the object-oriented programming. You will also learn about objects and classes, inheritance, polymorphism, abstraction and encapsulation. And, at the end of this chapter you will gain knowledge about the constructor and destructor as well.

5.2 Object-Oriented Programming

Object Oriented Programming (OOP) approach is introduced to fix the gaps encountered in procedural language. In this approach, the focus is on data that needs to be manipulated rather than the logic to manipulate that data. OOP is actually a programming paradigm that is based on the concept of objects which represents data. The basic idea of OOP is keeping data and its function together and preventing it from modification by an outside function. As the name implies, OOP revolves around objects, whose behaviour is described by the classes (instance of object).

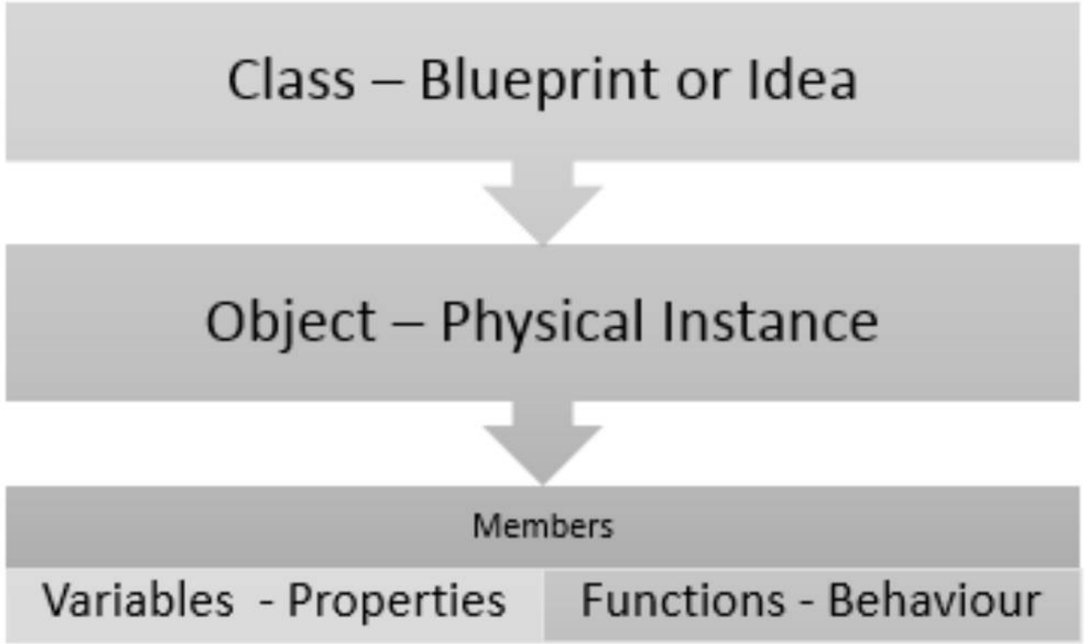
The primary purpose of OOP is to make an object, which has some properties and associated methods, and a group of objects interact together to complete certain programming tasks or operations.

Some important features of OOP are as follows:

- It mainly provides importance to data rather than procedure.
- The programs are divided into objects which can be characterised using data structures — array, union, structures, etc.
- The data of an object can be combined together with functions in the data structure.
- There is no possibility of directly accessing the private data by external function as the data is hidden. It is, however, possible by using pointers and friend function.
- Objects may communicate with each other through functions.
- You can easily add new data and functions whenever necessary.
- It follows the bottom-up approach in program designing.
- The complex problems will be solved easily as it divides those complex problems into smaller sets by using objects.

Here are the four important concepts in OOP:

Class – Blueprint or Idea


Class – Blueprint or Idea
Object – Physical Instance
Members
Variables - Properties
Functions - Behaviour

5.3 Class

A class is a user-defined data type, which binds the data members and its function members in a single package. In other words, a class is specified as a blueprint that describes the behaviours or states for an object. For instance, human is a class that contains body parts and performs some actions. Classes are declared with the keyword class. Some important points on class are as follows:

- A class is a user-defined data type.
- Class name should start with an uppercase letter. The first letter of each word should be in uppercase if the name of the class has more than one word.
- Member functions of the class can be defined inside or outside the definition of the class.
- Classes include data members and member functions.
- Classes are the base of all the OOPs concepts — inheritance, abstraction, encapsulation and polymorphism.

Example 1: Defining a class in C++

```
class Human
{
    string name;
    int age;
    char gender;
    public:
```

```
class Human
{
    string name;
    int age;
    char gender;
    public:
    void display(){
        cout<<"Hello "<<name<<". You are a "<<age<<" years old "<<gender<<endl;
    }
}
```

In the above example, a class named Human is defined in C++. It is binding the properties or state of human, such as name, age and gender, along with the function used to display human properties, together inside the container like structure called as class. The properties or variables are called as data members and the methods are called as member functions.

5.4 Object

An object is a basic unit of OOP which contains data variables and code (member function) to manipulate it. It can represent a place, person, data table or an account. For example, consider a dog has features, such as colour, name and breed, and its behaviour, such as bark, eat, etc., are its functions.

Objects are instances of class and are assigned certain space in memory. An object can interact with other objects without knowing their details. Each object has different data variables. The initialisation of objects is through special class member function named constructor. Each time the object is out of its scope, a destructor (another special class member function) is called to release the memory held by the object. Special class functions called constructors are used for initialising objects. On the other hand, for releasing the memory reserved by the object, another special class member function called destructor is called every time the object goes out of scope.

For creating an object in Java, you need to:

1. **Declare an object**: We can declare an object by using the class name and the object name. The syntax is:

```
class ClassName {

      DataType InstanceVariable;

DataType Method{

// method code

}
```

```
}

}
```

2. **Instantiating an object**: Instantiation of an object is done with the help of the "new" operator. When we instantiate an object, a space in the memory is acquired for that object. The syntax is:

```
object name = new class name();
```

Thus, the syntax to create an object is:

```
class name object name = new class name();
```

Multiple objects of a class can be created and these objects have separate copies of data members. For the class defined in example 1 its implementation is shown in Figure 1:

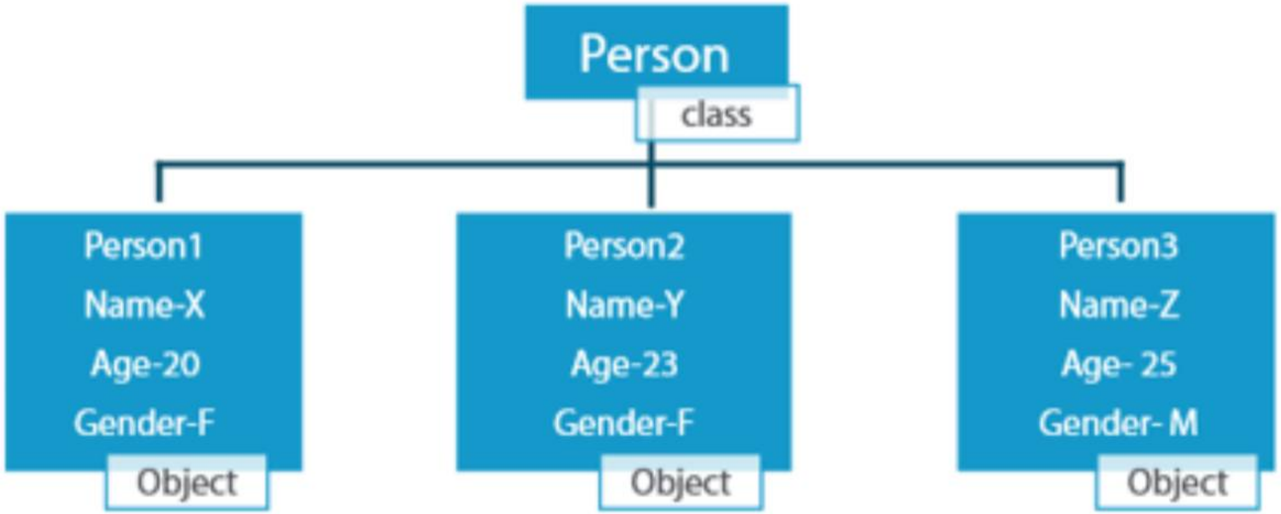


Figure 1: Multiple Object Creation in Class

5.5 Members

There are two types of members declared in the class and present in the objects as seen in above examples:

1. **Data members** – These are the member variables or state of the object. Each object has its own copy of data member variables.
2. **Member functions** – These are the behavioural aspects of the class or object. Usually the action elements are represented as the functions are methods.

Example 2: Creating Member

```
class Human
{
    string name;
    int age;
    char gender;
    public:
    void Human(){
            cout<<"Enter your name, gender (M/F) and age :\n";
            cin>>name>>gender>>age;
    }
```

```
        }
        void display(){
                cout<<"Hello "<<name<<". You are a "<<age<<" years old "<<gender<<endl;
        }
    }
int main()
{
        Human person1;
        person1.display();
}
```

Working: Let us identify the elements of this code,

Class: Human

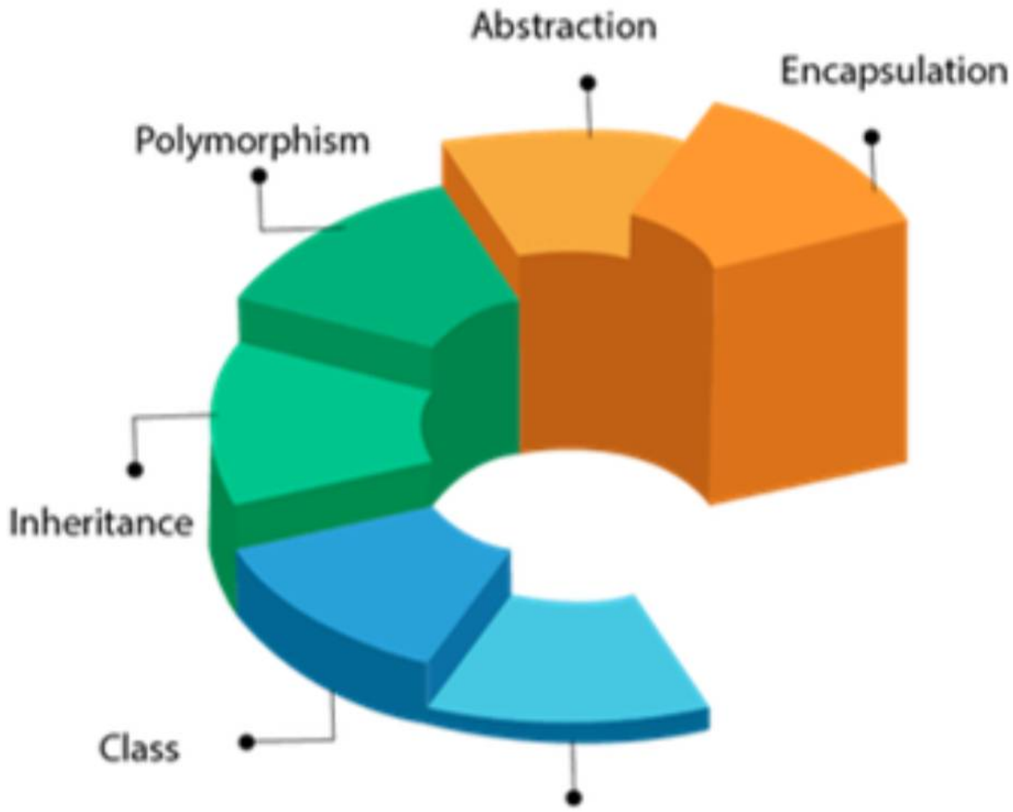
Object: person1

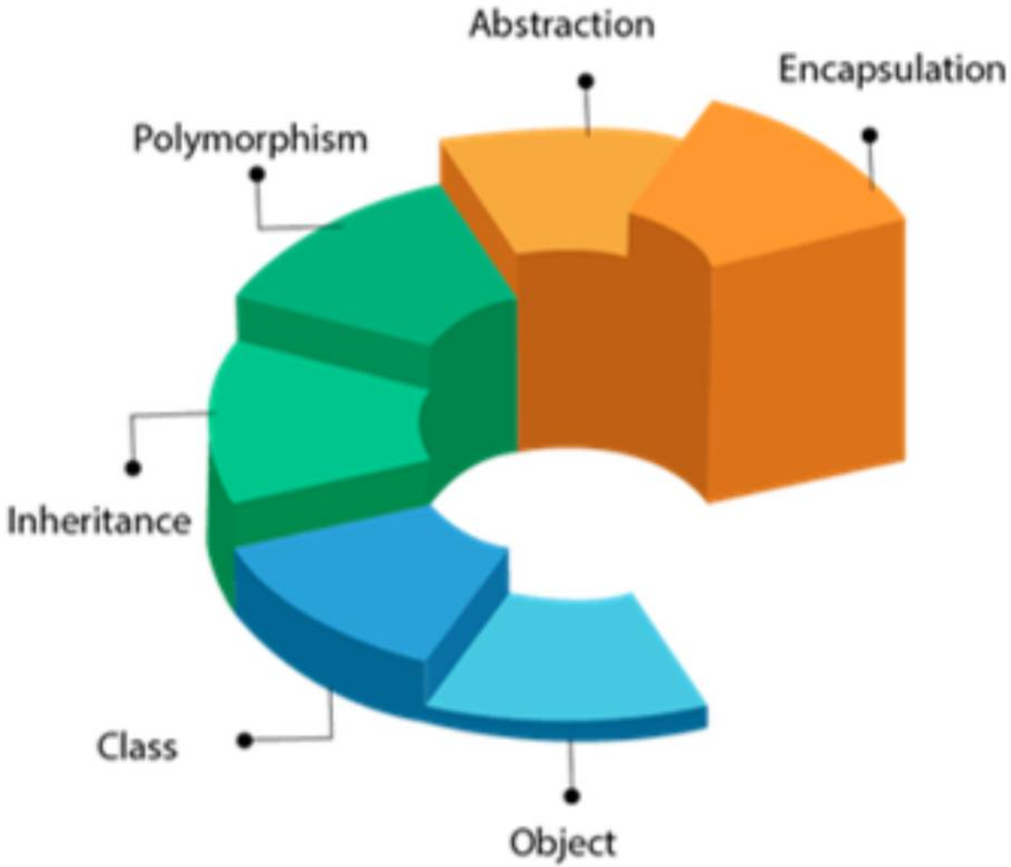
Data members(variables): name, gender, age

Member function: Human(), display()

5.6 The Four Pillars of OOP

Object oriented programming has four main concepts as shown below:




Abstraction
Encapsulation
Polymorphism
Inheritance
Class
Object

5.7 Abstraction

Data abstraction refers to displaying only the required information to the outside world without showing the background details, i.e. showing only the essential features of the application and hiding the details. For example, a database system hides the process of storing, creating and maintaining the data. Similarly, different methods are provided by C++ classes to the outside world but the internal details regarding the methods and data involved are not disclosed to them.

Let's take another real-life example to explain data abstraction. Consider a television, which you can switch on and off, change the channels, adjust the volume, colour and brightness, and add external components, such as speakers and DVD players. However, the internal working principle of the TV is not known to you. You have no idea about how signals are received by the television over the air or through a cable, how these signals are translated and how the images are displayed on the screen.

In case of OOPs, classes offer high level of data abstraction by providing public methods to the outside world. These methods allow users to do as many manipulations on object data. When defining a class, we specify both data members and member functions in the program. But, the member functions and built-in data types are hidden while using the object. This is referred as data abstraction. For instance, we have predefined classes, namely ostream and istream. We can use their objects, cout and cin, without seeing the internal code and function specifications.

5.7.1 Access Labels Enforce Abstraction

Access labels/access specifiers can be used in C++ to define the access control rules of the class. While defining the class, we can put access labels to the class and class members. It is not mandatory to put access label. You can put none or multiple access labels in the class. There are three keywords used to define access specifiers, namely public, private and protected.

- **Public Access Specifier**: By declaring public access specifier with class, all the class members will be available to everyone. Other classes can access the public data members and member functions of a class. Example 3 shows the use of public access specifier:

Example 3: Public Access Specifier

- **Public Access Specifier**: By declaring public access specifier with class, all the class members will be available to everyone. Other classes can access the public data members and member functions of a class. Example 3 shows the use of public access specifier:

Example 3: Public Access Specifier

```
class MyClass
{
      public:           // Public Access Specifier
      int a;            // Data Member Declaration
      void show();      // Member Function Declaration
}
int main()
{
      MyClass obj;
      cout<<obj.a;      //Since 'a' and 'show()' are public they
```

```
        cout<<obj.a;      //Since 'a' and 'show()' are public they
        obj.show();       //can be accessed outside the class as
    }                     //well.
```

- **Private Access Specifier**: The class members declared with private access specifiers cannot be accessed by anyone outside that class. All the variables and member function are defined as private by default. Example 4 shows the use of private access specifier:

Example 4: Private Access Specifier

```
class MyClass
{
    Private:          // Private Access Specifier
    int a;            // Data Member Declaration
    void show();      // Member Function Declaration
}
int main()
```

```
int main()
{
      MyClass obj;
      cout<<obj.a;      //Since 'a' and 'show()' are now private
      obj.show();       //accessing them outside the class, will
}                       //raise an error.
```

- **Protected Access Specifier**: Class members defined as protected can be accessed by its subclass but cannot be accessed outside the class. Example 5 shows the use of protected access specifier:

Example 5: Protected Access Specifier

```
class MyClass
{
      protected:        // Protected Access Specifier
      int a;            // Data Member Declaration
```

```
        void show();     // Member Function Declaration
    }
    class MySubClass: public MyClass
    {
        cout<<a;         // Since 'a' and 'show()' are protected
        show();          //accessing them in child class is valid
    }
    int main()
    {
        MyClass obj;
```

```
        cout<<obj.a;     //Since 'a' and 'show' are now protected
        obj.show();      //accessing them outside the class, will
    }                    //raise an error.
```

Every access level is applicable to the member definition following the label. The effect of a particular level is until the access label is specified or till the time closing brace of the class is encountered.

- All parts of the program can access members defined with a public label. The public members can be used to define the data abstraction view of class members and member functions.
- The code that uses the class cannot use the members defined with a private label in that class. The private sections hide the implementation of code from outer functions.

Example 6 shows how to find out multiplication of two numbers using abstraction concept in C++:

Example 6: Abstraction Example in C++

```
#include <iostream>

#include <conio.h>

using namespace std;

class Result
```

```
{
    private: int a, b, c;
    public:  void multiply()
    {
        cout<<"Enter first number: ";
        cin>>a;
        cout<<"Enter second number: ";
        cin>>b;
        c= a*b;
        cout<<"Multiplication of two numbers is: "<<c<<endl;
    }
```

```
};

int main()

{

    Result mul;

    mul.multiply();

    getch();

    return 0;

}
```

Figure 2 displays the output of Example 6:

```
Enter first number: 5
Enter second number: 2
Multiplication of two numbers is: 10
```

Figure 2: Output of Example 6

In Example 6, a class Result is defined which has private data members, namely a, b and c as well as public function which is multiply(). The multiply() function is defined within the class but as it has public access label, it can be accessed by the main function as well. The variables a, b and c have private access, so they can be accessed within the class Result only. These variable are used by multiply() function, which is a member of the same class.

In main () function, only an object of class Result is created and this object is used to call multiply function via dot(.) operator (operator used to access members of an object). The fact that private data members a, b and c cannot be accessed by the main function represents the abstraction in Example 6. User is able to see the result of multiplication of two numbers without knowing about the internal data members.

5.7.2 Abstract Methods

If we want to make a method in a parent class and its body in the child class, then we make that method abstract. For making this method, we put the abstract keyword before the method name. We do not give the body of abstract method at the time of declaration because its body is defined in another class. Example 7 illustrates the concept of abstraction:

Example 7: Abstraction Example in Java

```
abstract class Vehicle{

      abstract void move();

}

class Car extends Vehicle{
```

```
        void move(){
                System.out.println("Car is moving");
        }
    }
    class AbstractionExample{
        public static void main(String args[]){
                Car C1 = new Car();


                C1.move();
        }
    }
```

Figure 3 displays the output of Example 7:

```
}
```

Figure 3 displays the output of Example 7:

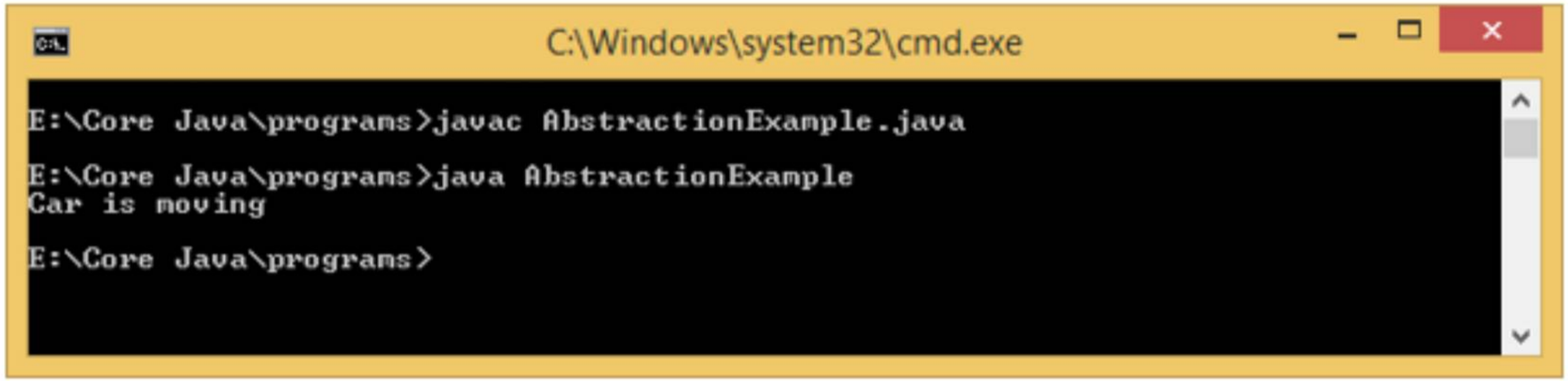


Figure 3: Output of Example 7

5.8 Encapsulation

Encapsulation is the process of binding together the data members and member functions that manipulate the data, thereby keeping the members safe from outside misuse. In other words, it is the process of hiding the data and functions into a single unit and protecting it from the outside world. Data encapsulation provides an important OOP concept of data hiding. C++ supports the properties of encapsulation and data hiding through the user-defined types called classes.

You might get confused in differentiating between abstraction and encapsulation. For these two concepts, we take an example of a mobile phone. The outer look of a mobile phone includes display screen and keypad buttons to dial numbers. The outer layout of design is called abstraction. The inner implementation of mobile phone includes how the display screen and keypad button are connected with each other. The inner layout implementation is referred to as encapsulation.

The features of data encapsulation are as follows:

- Whenever the implementation of the class changes, the interface remains the same.
- It reduces human errors.
- It makes application maintenance easier.
- It improves the understandability of the application.
- It provides enhanced security.

Example 8 shows how to set and get the value in C++:

Example 8: Encapsulation in C++

```
#include <iostream>

#include <conio.h>
```

```
using namespace std;

class Value

{

    private:        int a;

    public:       void set(int x)

    {

        a =x;

    }

    int get()

    {

        return a;

    }
```

```
};

int main()

{

    Value v;

    v.set(5);

    cout<<v.get();

    getch();

    return 0;

}
```

Figure 4 displays the output of Example 8:

```
5
```

Figure 4: Output of Example 8

In Example 8, a class Value is defined. As you can see, the data member (i.e. int a) and member functions (set and get) are defined within the class. Only set() and get() functions can access the value of private data member 'a'. In the main function, an object of the Value class is defined. This object is used to initialise member variable 'a' by calling the set() function and then displaying the value of 'a' by calling the get() function.

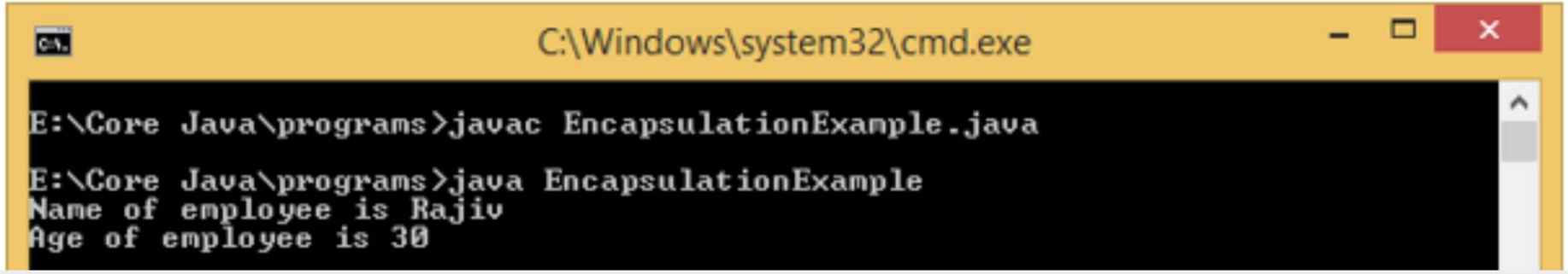
Example 9 illustrates the concept of encapsulation in Java:

```
class Employee{
    private String name;
    private int age;
    public int getAge(){
        return age;
    }
    public String getName(){
        return name;
    }
```

```
        public void setAge(int age){


                this.age = age;

        }

        public void setName(String name){

                this.name = name;

        }

}

class Encapsulation Example{

        public static void main(String args[]){

                Employee E1 = new Employee();

                E1.setName("Rajiv");
```

```
            E1.setName("Rajiv");

            E1.setAge(30);

            System.out.println("Name of employee is " + E1.getName());

            System.out.println("Age of employee is " + E1.getAge());


        }

}
```

Figure 5 displays the output of Example 9:



5.9 Inheritance

Inheritance is an important concept in OOP that allows us to define a new class from an existing class, thereby making it easier to create and maintain a software application. It provides an opportunity to reuse the code functionality, i.e. a way to reuse once written code again and again, and helps in reducing the code size and ensuring faster implementations.

The existing class from which the new class has inherited is called base class, super class or parent class. The new class that inherits the base class is called derived class, sub class or child class. The derived class can access and use all the functions of the base class.

Figure 6 shows Human as the parent class while Male and Female are child classes. All humans have two hands, two legs, one face and they can talk, eat, walk and run. Male and Female are child classes, so they have these states and behaviour in common. These features are inherited from their parent class, Human. Apart from these features, Male and Female can have some additional features of their own.

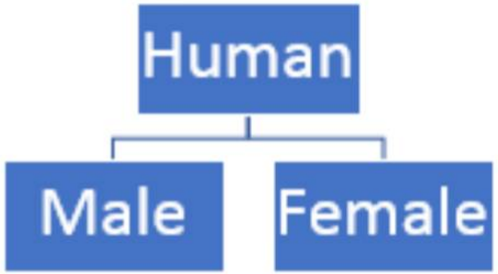


Figure 6: Example of Inheritance

Through inheritance, a programmer is prevented from unnecessary coding. He/she can use only those qualities or features of the base class, which are required to ease the functionality of new code. The derived class can inherit any attributes from its parent class to access both the data and methods, and later on, you can add any new data and methods

Figure 6: Example of Inheritance

Through inheritance, a programmer is prevented from unnecessary coding. He/she can use only those qualities or features of the base class, which are required to ease the functionality of new code. The derived class can inherit any attributes from its parent class to access both the data and methods, and later on, you can add any new data and methods whenever required.

There are five types of Inheritance:

- **Single Inheritance**: Only one class can be inherited in single inheritance. This means that there will be only one base class and one derived class. Figure 7 shows example of single inheritance:

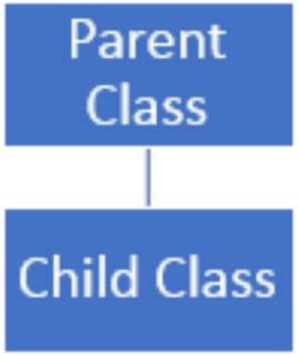


Figure 7: Single Inheritance

Example 10 demonstrates single inheritance:

Example 10: Single Inheritance in C++

```
#include <iostream>

#include <conio.h>

using namespace std;

// base class definition

class Human {

    public: Human()

    {

        cout << "I am a Human" << endl;

    }

};

// sub class derived from base class

class Male: public Human{};
```

```
// main function
int main()
{
  // object of sub class invoke the constructor of base
  class Male obj;
  getch();
  return 0;
}
```

Figure 8 displays the output of Example 10:

```
I am a Human
```

Figure 8: Output of Example 10

- **Multiple Inheritance**: When a class is inherited from more than one classes, it is referred to as multiple inheritance. This means there will be one derived class and more than one base class. Figure 9 shows the multiple inheritance:

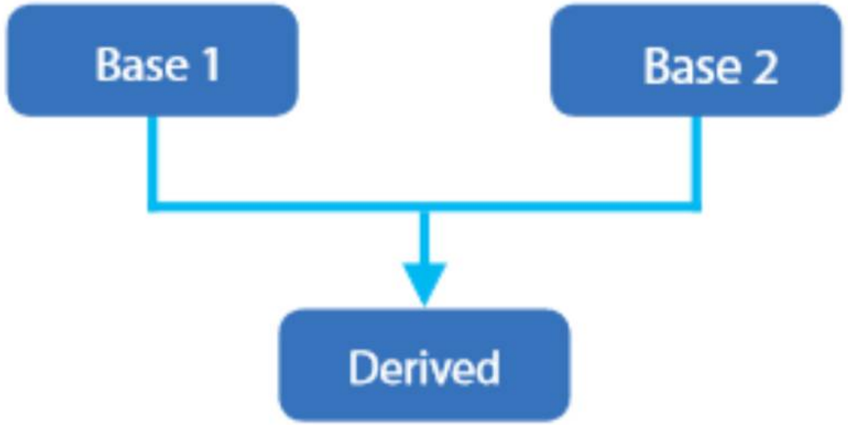


Figure 9: Multiple Inheritance

Example 11: Multiple Inheritance

```
#include <iostream>

#include <conio.h>

using namespace std;
```

```
//base class 1 definition
class Building {
    public:    Building()
    {
        cout << "I am in a building"<< endl;
    }
};
// base class 2 definition
class Hotel {
    public:    Hotel()
    {
        cout << "I am in a Hotel"<<endl;
```

```
    }

};

// sub class derived from two base classes

class Room: public Building, public Hotel{};

// main function

int main() {

    // object of sub class invoke the constructor of base classes

    Room obj;

    getch();

    return 0;

}
```

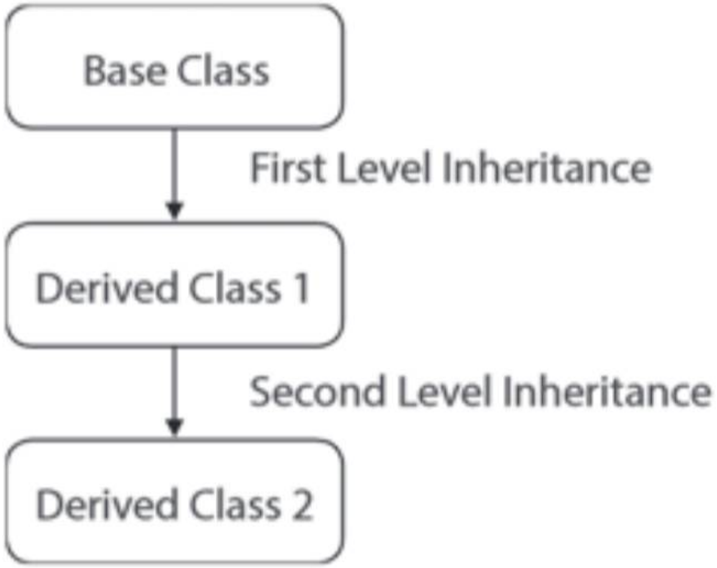
Figure 10 displays the output of Example 11:

Figure 10 displays the output of Example 11:

```
I am in a building
I am in a Hotel
```

Figure 10: Output of Example 11

- **Multilevel Inheritance**: One derived class is inherited from another derived class in multilevel inheritance. Figure 11 shows the multilevel inheritance:



Example 12 demonstrates multilevel inheritance in C++:

Example 12: Multilevel Inheritance

```
#include <iostream>

#include <conio.h>

using namespace std;

class Building {

    public:    Building()

    {

        cout << "I am in a building"<< endl;

    }

};

// sub class 1 derived from base class
```

```
class Hotel: public Building {
    public:    Hotel()
    {
        cout << "I am in a Hotel"<<endl;
    }
};
// sub class 2 derived from derived classes
class Room:  public Hotel {
    public:    Room()
    {
        cout << "I am in a Room"<<endl;
    }
```

```
    }
};
// main function
int main(){
    // object of sub class 2 invoke the constructor of base class,
sub class 1 and sub class 2


      Room obj;
      getch();
      return 0;
  }
```

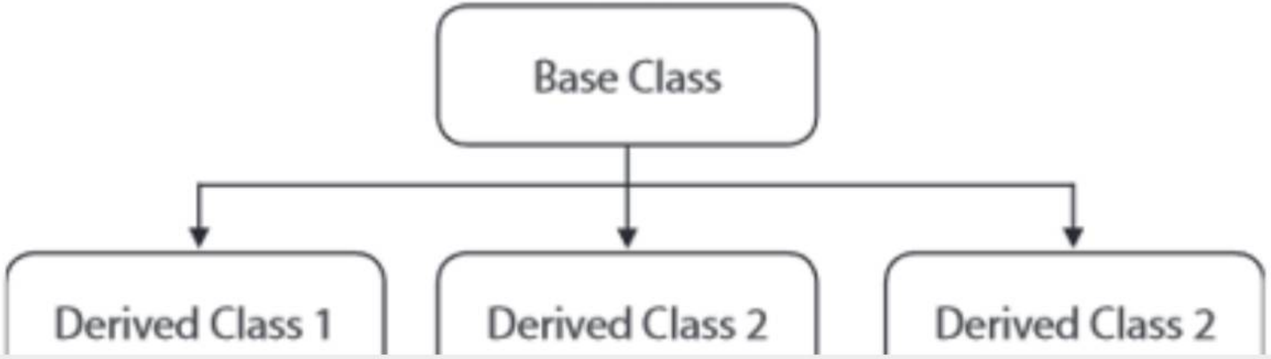
Figure 12 displays the output of Example 12:

```
I am in a building
I am in a Hotel
I am in a Room
```

Figure 12: Output of Example 12

In Example 12, Building is the base class and Building class, Hotel is the class derived from Building class and Room is the class derived from Hotel class. On invoking the object of Room class, first the constructor of Building class is invoked, then Hotel class constructor is invoked and finally Room class constructor is invoked and corresponding messages are displayed on the screen.

- **Hierarchical Inheritance**: Multiple derived classes can be inherited from a single base class, that is more than one derived class is created from a single base class. Figure 13 shows the hierarchical inheritance:



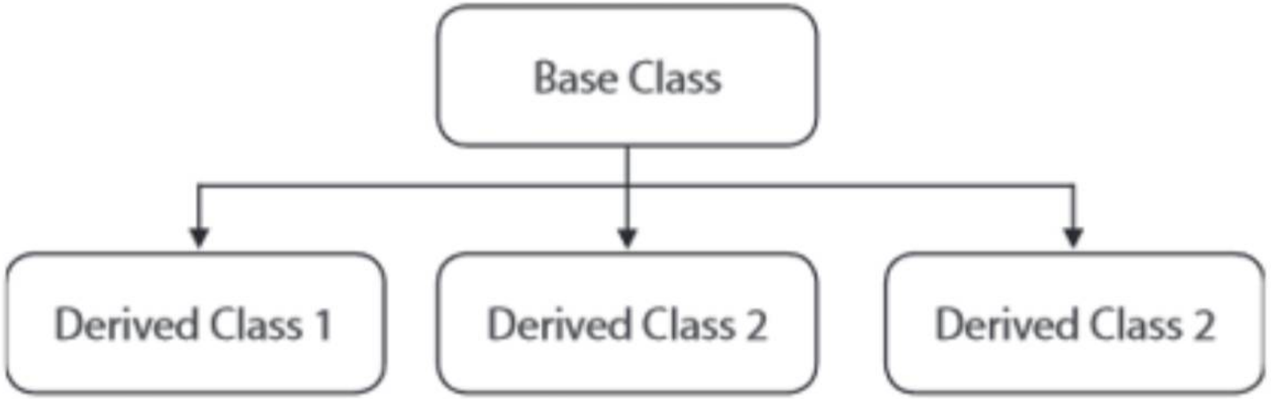


Figure 13: Hierarchical Inheritance

Example 13 demonstrates hierarchical inheritance:

Example 13: Hierarchical Inheritance

```
#include <iostream>

#include <conio.h>

using namespace std;

// base class
```

```
class Building{


    public:    Building()
    {
        cout << "I am in a building"<< endl;
    }
};
// first sub class
class Hotel: public Building{
    public:    Hotel()    {
    }
};
```

```
// second sub class
class Resort:  public Building{
    public:    Resort()    {
    }
};
// main function
    // object of sub class invoke the constructor of base class
    Hotel hobj;
    Resort robj;
    getch();
    return 0;
}
```

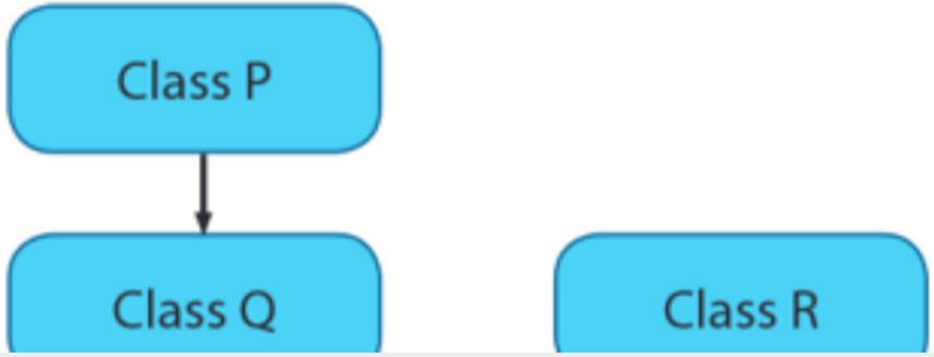
```
}
```

Figure 14 displays the output of Example 13:

```
I am in a building
I am in a building
```

Figure 14: Output of Example 13

- **Hybrid Inheritance**: When more than one type of inheritance is implemented together, it is called hybrid inheritance. It is also called virtual inheritance. Figure 15 shows the hybrid inheritance combining hierarchical inheritance and multilevel inheritance:



- **Hybrid Inheritance**: When more than one type of inheritance is implemented together, it is called hybrid inheritance. It is also called virtual inheritance. Figure 15 shows the hybrid inheritance combining hierarchical inheritance and multilevel inheritance:

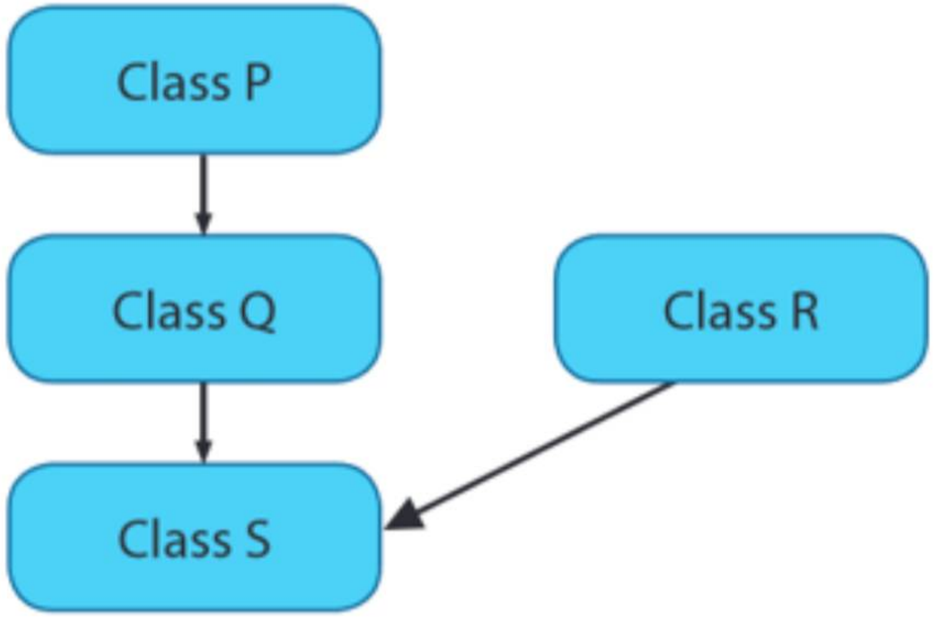


Figure 15: Hybrid Inheritance

Example 14: Hybrid Inheritance

```
#include<iostream>
```

```
#include<iostream>

#include<conio.h>

using namespace std;

int p,q,r,s,t;

class P{

    protected:

    public: void getpq()

    {

        cout<<"\n Enter the value of p:";

        cin>>p;

        cout<<"\n Enter the value of q:";

        cin>>q;
```

```
        cin>>q;
    }
};
class Q: public P {
    protected:

     public: void getr()
     {
         cout<<"\n Enter the value of r:";
         cin>>r;
     }
 };
class R{
```

```
class R{
    protected:
    public: void gets()
    {
        cout<<"\n Enter the value of s:";
        cin>>s;
    }
};
class S:public Q,public R{
    protected:
   public: void sum() {
       getpq();
```

```
        getpq();
        getr();
        gets();
        t=p+q+r+s;
        cout<<"\n Sum is :"<<t;
    }
};
int main(){
    S s1;
    s1.sum();
    getch();

}
```

```
}
```

Figure 16 displays the output of Example 14:

```
Enter the value of p:10

Enter the value of q:20

Enter the value of r:30

Enter the value of s:40

Sum is :100
```

Figure 16: Output of Example 14

5.10 Polymorphism

The word “polymorphism” is a combination of the words “poly”, which means “many”, and “morph”, which means “forms”, in short, many forms. In other words, the ability to take more than one form is referred to as polymorphism. In technical terms, Polymorphism is the capability of using an operator or a function in different ways. A single function or an operator can function in many ways on different instances depending upon the usage.

Polymorphism can be used in the following two ways:

- **Static (Compile Time) Polymorphism**: Static polymorphism implies an entity existing in different physical forms concurrently, i.e. the response to the member function is determined at the time of compilation. The member functions are predefined to behave as per the definition based on the type, number and sequence of arguments. It is also known as static binding or early binding. Static overloading can be implemented using operator overloading and function overloading.
    - **Operator Overloading**: It is a technique to define two or more meanings of an existing operator. For example, “+” operator adds two numbers as well as concatenate two strings, “*” used for multiplication as well as “value at operator”.

Example 15: Operator Overloading

```
#include <iostream>

using namespace std;

class Complex{

        private:
```

```
        int real,img;
    public:
        Complex(int r=0,int i=0){
            real=r;
            img=i;
        }
        void display(){
            cout<<real<<" + i "<<img;
        }
        Complex operator + (const Complex &c) {
            Complex r;
            r.real = real + c.real;
```

```
                r.img = img + c.img;
                return r;

        }
};


int main() {
        int i=0;
        Complex c1(4,5);
        Complex c2(6,7);
        c1.display();
        cout<<endl;
```

```
        c2.display();
        Complex c3=c1+c2;          //Operator '+' is overloaded for the complex objects.
        cout<<endl;
        c3.display();
        return 0;
}
```

Figure 17 displays the output of Example 15:

```
4 + i 5
6 + i 7
10 + i 12
```

Figure 17: Output of Example 15

- **Function Overloading**: This means the same function name exists numerous times in the same class with different types of parameters, different order of parameters or different number of parameters. For example, add(int a, int b), add(int a, int b, int c) and add(double a, double b) have same name but operates differently as shown in Example 16:

- **Function Overloading**: This means the same function name exists numerous times in the same class with different types of parameters, different order of parameters or different number of parameters. For example, add(int a, int b), add(int a, int b, int c) and add(double a, double b) have same name but operates differently as shown in Example 16:

Example 16: Function Overloading

```
#include<iostream>

#include<conio.h>

using namespace std;

int add(int a, int b){

    cout << a+b <<endl;

}

int add(int a, int b, int c){

    cout << a+b+c <<endl;

}

double add(double a, double b){
```

```
double add(double a, double b){
    cout << a+b <<endl;
}
int main(){

        add(5,10);
        add(5,10,15);
        add(5.5,4.5);
        getch();
    }
```

Figure 18 displays the output of Example 16:

```
15
```

```
    add(5,10,15);

    add(5.5,4.5);

    getch();

}
```

Figure 18 displays the output of Example 16:

```
15
30
10
```

Figure 18: Output of Example 16

In Example 16, add(int a, int b), add(int a, int b, int c) and add(double a, double b) have same name but behave differently.

- **Dynamic (Run-time) Polymorphism**: Dynamic polymorphism implies an entity changing its form as per the circumstances, i.e. choosing the member function during the program execution. When a function exists in multiple forms whose calls are resolved dynamically upon program execution, it is said to exhibit dynamic polymorphism. It is also called dynamic binding or late binding.

In Example 16, add(int a, int b), add(int a, int b, int c) and add(double a, double b) have same name but behave differently.

- **Dynamic (Run-time) Polymorphism**: Dynamic polymorphism implies an entity changing its form as per the circumstances, i.e. choosing the member function during the program execution. When a function exists in multiple forms whose calls are resolved dynamically upon program execution, it is said to exhibit dynamic polymorphism. It is also called dynamic binding or late binding.

While using dynamic polymorphism, it is possible to have same parameter in subclass as in its super classes with the same name. For example, method overriding — defining a method in both parent class and child class with similar signature, name or parameters.

Example 17 demonstrates dynamic polymorphism:

Example 17 demonstrates dynamic polymorphism:

Example 17: Dynamic Polymorphism

```
#include <iostream>

#include <conio.h>

using namespace std;

class Team1 {

    public:  void play(){

        cout<<"Playing Cricket!!!"<<endl;
```

```
     cout<<"Playing Cricket!!!"<<endl;
 }

};
class Team2: public Team1    {
    public:   void play()     {
        cout<<"Playing Football!!!"<<endl;
    }
};
int main(void) {
    Team1 tm1;
    Team2 tm2;
```

```
    tm1.play();

    tm2.play();

    getch();

    return 0;

}
```

Figure 19 displays the output of Example 17:

```
Playing Cricket!!!
Playing Football!!!
```

Figure 19: Output of Example 17

As we can see in Example 17, two functions with similar names are defined in both, base class (Team 1) and derived class (Team 2). Both the functions can be called by using the object of each class. As the play() function is identical in both the classes in the program. This is the case of method overriding. It is not possible to know that which function will execute during compilation and will be resolved at runtime. Object of the class in which function is defined decides which function to call.

5.10.1 Method Overloading

Method overloading is used to achieve polymorphism and increase the readability of a program. Overloading a method means creating two or more methods with the same name. We can achieve method overloading in the following two ways:

1. By changing the number of argument of methods
2. By changing the data type of argument of methods

Example 18 illustrates the concept of method overloading in Java:

Example 18: Method Overloading

```
class Addition{
      int add(int a, int b){
            return a+b;
      }
      int add(int a, int b, int c){
            return a+b+c;
```

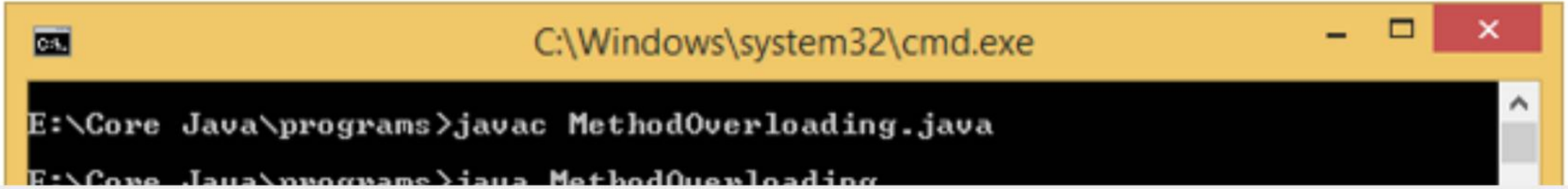
```
        }
        double add(double a, double b, double c){
            return a+b+c;
        }
    }
    class MethodOverloading{
        public static void main(String args[]){
            Addition A1 = new Addition();
            int x;
            double y;
            x = A1.add(12,13);
            System.out.println("The value of addition = " + x);
```

```
        x = A1.add(12,13,14);

        System.out.println("The value of addition = " + x);

        y = A1.add(22.5,23.5,24.5);

        System.out.println("The value of addition = " + y);

    }

}
```

In Example 18, we are creating the add method and overloading it twice. The first add method takes two integer arguments and the second add method takes three integer arguments. The overloading is done by changing the number of argument. Next, we are creating another add method that also takes three arguments, but in double type. Therefore, here overloading is done by changing the data type of the method argument.

Figure 20 displays the output of Example 18:

Figure 20 displays the output of Example 18:



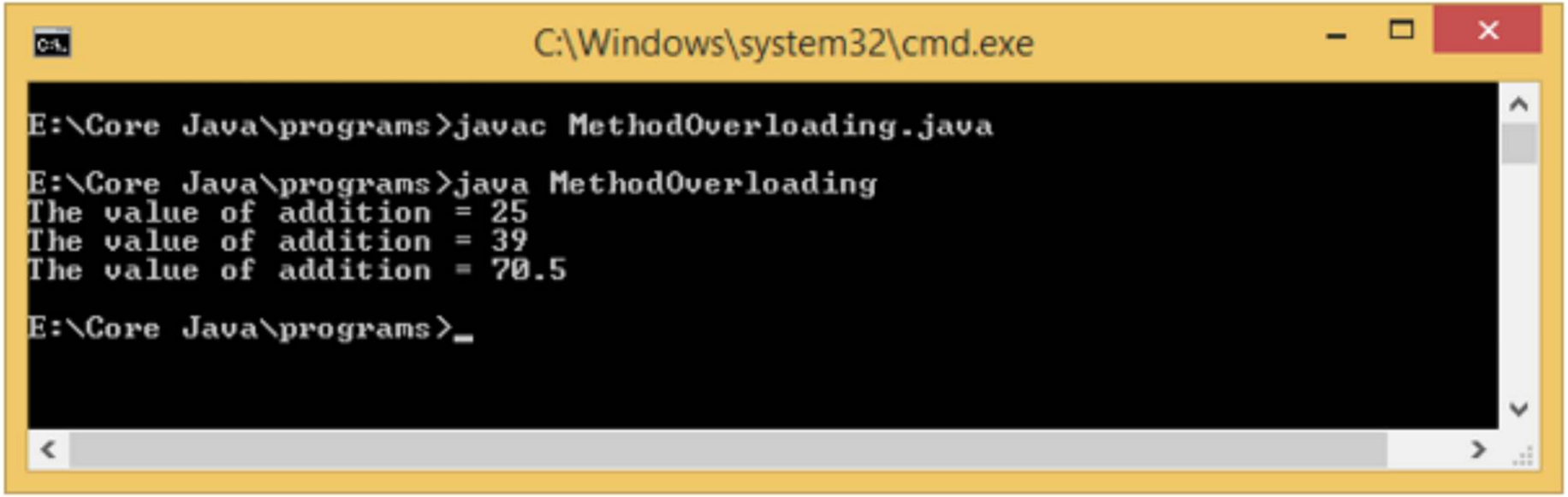


Figure 20: Output of Example 18

5.10.2 Method Overriding

In inheritance, the child class inherits the method of the parent class. If the child class needs to define the body of the method that is already in the parent class, then the child class uses the overriding concept. Overriding a method means creating a method in the child class with the same name and parameter of the method that is present in the parent class. When we override a method in the child class, the parent class method hides. For accessing the parent class method, we can use the super keyword.

Example 19 illustrates the method overriding concept:

```
class Vehicle{
      void run(){
            System.out.println("Vehicle is running");
      }
      void color(){
            System.out.println("The color of vehicle is black");
      }
}
class Car extends Vehicle{
      void run(){



            System.out.println("Car is running");
```

```
            System.out.println("Car is running");
        }
        void color(){
            super.color();
        }
    }
    class MethodOverriding{
        public static void main(String args[]){
            Car C1 = new Car();


            C1.run();
            C1.color();
```

```
        }

}
```

Figure 21 displays the output of Example 19:

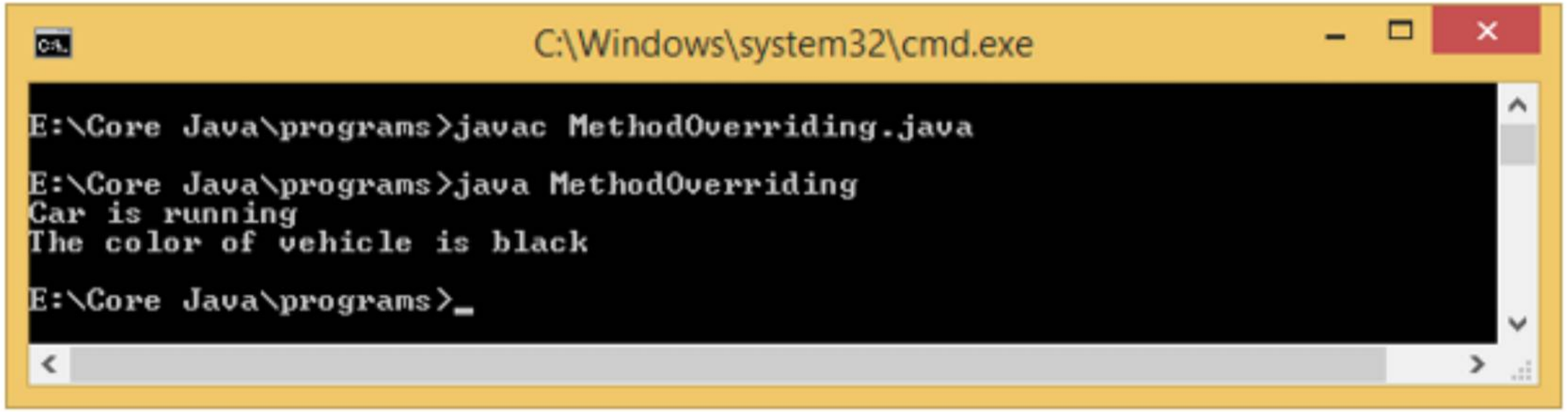


Figure 21: Output of Example 19

Function overloading	Function overriding

Function overloading	Function overriding
Function overloading has two or more functions with same name but different signature, i.e. different number and/or type of arguments.	Function overriding requires two identical declared functions in parent and child classes. However, their definition or implementations may vary.
Function overloading requires the overloading functions to be in the same scope.	Function overriding requires functions to be present in hierarchical classes.
Function overloading can take place in procedural as well as object-oriented approach, but inheritance is not required.	Function overriding requires inheritance in object oriented implementation. It cannot be implemented in procedural approach.
The binding of function to the function call	The binding of unction to the function call

Function overloading can take place in procedural as well as object-oriented approach, but inheritance is not required.	Function overriding requires inheritance in object oriented implementation. It cannot be implemented in procedural approach.
The binding of function to the function call takes place at compile time.	The binding of unction to the function call takes place at run time.
Virtual functions and dynamic binding cannot be implemented using function overloading.	Virtual functions and dynamic binding require function overriding.

5.11 Initialisation and Finalisation

The lifetime of an object includes the interval during which it had occupied the space and was holding the data. Most object-oriented languages provide a mechanism that helps in initialising the object automatically at the beginning of its lifetime. This mechanism is known as a constructor. A constructor does not allocate a space, it is used to initialise the space that has been allocated. Some languages also provide a destructor to finalise an object automatically. Let us discuss these two in the following section.

5.11.1 Constructor

A constructor is a special member function of a class which is called automatically whenever you create a new object of a particular class. The name of the constructor is exactly the same as the name of the class. Moreover, it does not have any return type. It is used in setting initial values for the data members of a class. Syntax of the constructor in C++ is as follows:

```
Class classname  // class {

private: Data member and member functions

protected: Data member and member functions

public: Data member and member functions

Classname(parameters) //constructor {}

};
```

Constructors are like normal functions with a difference that you do not need to write any statement to call constructors as we do in case of normal functions. Defining the object of the

Constructors are like normal functions with a difference that you do not need to write any statement to call constructors as we do in case of normal functions. Defining the object of the class invokes the constructor automatically. As the name suggests, constructors construct the value of data members of a class.

They can be defined inside or outside the class. For defining them outside the class, scope resolution operator is used. The main use of constructor is to initialise the data members with user-defined values instead of default values. To define constructor outside the class, write:

```
classname :: classname (parameters) {

}
```

Example 20 demonstrates how to add two numbers using constructor:

Example 20: Adding Numbers using Constructor

```
#include<iostream>

#include<conio.h>

using namespace std;

class add_func{

    int x,y,z;

    public:      add_func()        {
```

```
    public:      add_func()        {
        x=20;
        y=30;
        z=x+y;
        cout<<"Total value is:"<<z;
    }
};
int main(){
    add_func a;

     getch();
 }
```

Figure 22 displays the output of Example 20:

```
Total value is:50
```

Figure 22: Output of Example 20

5.11.2 Constructor Types

Constructors are special class functions used for initialising objects and are available in three main types. Every type of constructor has a different role. The different types of constructors used in C++ are as follows:

- **Default Constructor**: The default constructor is the one that can be defined with an empty parameter list. If no constructor is defined in a program, the compiler automatically provides a default constructor that initialises the object's data members with dummy values. The default constructor also reserves the memory for the object. The default constructor defined by the user can have a body but the default constructor defined by the compiler does not have a body. The syntax of the default constructor is:

class_name ( ){

. . .

. . .

-

- **Parameterised Constructor**: When parameters are passed to a constructor, they are called parameterised constructors. A parameterised constructor takes the parameters

- **Parameterised Constructor**: When parameters are passed to a constructor, they are called parameterised constructors. A parameterised constructor takes the parameters which provide different values to the data members by using values as arguments. The parameterised constructor is used in those situations where you want the data members of an object to be initialised by the values provided by the user at the time of object creation. The limitation of a default constructor is that the data members of every object are initialised with same set of values. However, parameterised constructor enables you to initialise data members of different objects with different values. It is useful in situations where you do not want every object to contain the same set of values for its data members. The syntax of a parameterised constructor is:

class_name(Argument_List) {

. . .

. . .

-

- **Copy Constructor:** A copy constructor copies the member values of one object to a newly created object, i.e. it creates a new object as a copy of an existing object. The copy constructor is used to:
    - Initialise one object by using another object of the same type.
    - Copy an object to pass it as an argument to a function.
    - Copy an object to return it from a function.

If the class is not provided with a user-defined constructor, the compiler will automatically supply a copy constructor. The copy constructor defined by compiler do only member-by-member copy operations. Default copy constructors make only shallow copies. The data members are copied irrespective of their types. Syntax of a copy constructor is:

class_name(const class_name &obj) {

. . .

. . .

member copy operations. Default copy constructors make only shallow copies. The data members are copied irrespective of their types. Syntax of a copy constructor is:

class_name(const class_name &obj) {

. . .

. . .

5.11.3 Overloading Constructors

You can overload the constructors in the similar way as you overload other member functions. Overloaded constructors will have the same name (name of the class), but different number of arguments. You can call the specific constructor whose parameters match the arguments. A program that has a default constructor as well as parameterised constructor is a common example of constructor overloading. Compiler does not provide any default constructor if user defines a constructor explicitly.

Example 21: C++ example to show constructor overloading

```
#include <iostream>

using namespace std;

class MyAddition {

        float res ;
```

```
public:
MyAddition () {

        int x, y;
        cout<<"Enter two numbers :\n" ;
        cin>>x>>y ;
        res = x + y ;
}
MyAddition (int a, int b) {
        res = a + b ;
}
MyAddition (int a, int b, int c) {
```

```
        MyAddition (int a, int b, int c) {
                res=a + b + c ;
        }
        disp_res()
        {
        cout<<endl<<"The result of addition is: "<<res;
        }
};

 int main()
 {
         MyAddition add1;                // This calls the first constructor
```

```
        add1.disp_res();

        MyAddition add2 ( 2, 3);        // This calls the second constructor

        add2.disp_res();

        MyAddition add3( 2, 3 ,4);      // This calls the third constructor

        add3.disp_res();

}
```

Example 21 when executed generates following output as shown in Figure 23 below.

```
Enter two numbers :
5
6

The result of addition is: 11
The result of addition is: 5
The result of addition is: 9
--------------------------------
```

<H2>Destructors

A destructor works opposite to a constructor. It is executed whenever an object of its class goes out of scope. When we create an object, a particular memory is allocated to it. If we will not destroy that object, the memory will remain occupied by the unused object. Destructor performs this task by releasing the memory occupied by an unused object. It is also executed whenever the delete expression is applied to a pointer to the object of that class. Some important points on destructors are as follows:

- Destructor name should be exactly the same as the name of the class, but it is preceded by the tilde sign (~).
- Destructor does not have any return type.
- It does not take any parameters.
- It frees the memory of an object whenever it goes out of scope.
- Destructors cannot be overloaded.
- Destructors can be virtual.

The syntax of destructor is:

```
~class_name(){

. . .

. . .

}
```

Example 22 demonstrates the use of destructors:

```
#include<iostream>

#include<conio.h>

#include<dos.h>

using namespace std;

class add_func{

    public:  int x,y,z;

    add_func(){

        x=15;

        y=25;

        z=x+y;

        cout<<"Total value is:"<<z<<endl;
```

```
    }
    ~add_func(){
        cout<<"calling destructor"<<endl;
        delay(2000);
        // to delay the execution of program for specified time
    }
};

int main(){
    add_func a;
    cout<<"calling main function"<<endl;
    getch();
}
```

```
    getch();
}
```

Figure 24 displays the output of Example 22:

```
Total value is:40
calling main function
calling destructor
```

Figure 24: Output of Example 22

5.12 Dynamic Binding

Simula, C++, and C# use static method binding by default where the programmer can specify the particular methods that should use dynamic binding by labelling them as virtual. These methods are called for the appropriate implementation at run time, based on the class of the object, rather than the type of the reference.

In C++, a function is declared using the keyword “virtual”, a virtual function is a function which can be defined within a base class and can be redefined in a derived class. By making a function virtual, we tell the compiler that static linkage is not required from this function. This means we want the function to be called at runtime for only those objects for which it is called. Syntax for virtual function is as follows:

```
virtual return_type function_name(){

. . .

. . .

}
```

Example 23 demonstrates the use of virtual function:

Example 23: Use of Virtual Function

```
#include <iostream>

using namespace std;
```

```
using namespace std;
class shape{                                //Base class
      public:
      virtual void area(){                  //Virtual overridden function
            cout<<"Area has units squared.";
      }
};
class cube:public shape{                    //Derived class
      int side;
      public:
            cube(){
              cout<<"Side? ";
```

```
            cin>>side;



        }
        void area(){                    //Overriding function
            cout<<"The surface area is "<<(6*side*side);
        }
};
class Cylinder:public shape{            //Derived class
    int r,len;
    public:
        Cylinder(){
            cout<<"r and l";
```

```
            cout<<"r and l";
            cin>>r>>len;
        }
        void area(){                    //Overriding function
            cout<<"The surface area is "<<(2*3.14*r*(r+len));
        }
};

int main() {
    shape *p;                           //Base class pointer
    shape s;                            //Base class object
    cube c;                             //Derived class object
    Cylinder o;                         //Derived class object
```

```
    cout<<endl;
    p=&s;
    p->area();                          //Call to base function
    cout<<endl;
    p=&c;
    p->area();                          //Call to derived function

    cout<<endl;
    p=&o;
    p->area();                          //Call to derived function
    return 0;
}
```

Figure 25 below shows the output for Example 23:

```
Side? 5
r and l
10
2

Area has units squared.
The surface area is 150
The surface area is 753.6
--------------------------------
```

Figure 25: Output of Example 23

If the keyword 'virtual' is removed from Example 23, the following would be the output.

```
Side? 5
r and l
10
2
```

If the keyword 'virtual' is removed from Example 23, the following would be the output.

```
Side? 5
r and l
10
2

Area has units squared.
Area has units squared.
Area has units squared.
--------------------------------
```

Figure 26: Output after removing the keyword "virtual"

5.13 Recap and Questions

5.13.1 Recap

Let us quickly recapitulate the following discussion points:

- An object is the fundamental unit of OOP which includes states and behaviours.
- A class is specified as a blueprint that describes the behaviours/states for an object.
- The starting letter for any class name should be in uppercase.
- Data abstraction is a programming technique that relies on the separation of interface and implementation.
- Data encapsulation provides an important OOP concept of data hiding.
- Inheritance is an important concept in OOP that allows us to define a new class from an existing class, thereby making it easier to create and maintain an application.
- Inheritance is of five types, namely single inheritance, multiple inheritance, multilevel inheritance, hierarchical inheritance and hybrid inheritance.
- A virtual function is a function declared using the keyword “virtual”, which can be defined within a base class and can be redefined in a derived class.

5.13.2 Test Your Skills- Multiple Choice Questions

1. Which of the following keywords is used to prevent any method from overriding?

a. Static
b. Const
c. Final

1. Which of the following keywords is used to prevent any method from overriding?

a. Static
b. Const
c. Final
d. Abstract

2. Which of the following define multiple inheritance?

a. One child class inheriting from more than one super classes.
b. More than one child classes inheriting from one super class.
c. More than one child classes inheriting from more than one super classes.
d. One child class inheriting from one super classes.

3. Which of the following is the correct declaration of an object of class Cuboid?

a. Cuboid C1 = new Cuboid();
b. Cuboid C1 = new Cuboid;
c. C1 = new Cuboid();
d. new Cuboid C1;

4. Super keyword is used to

a. refer an immediate child class.
b. refer an immediate parent class.
c. refer current class object.
d. refer static member of parent class.

4. Super keyword is used to

a. refer an immediate child class.
b. refer an immediate parent class.
c. refer current class object.
d. refer static member of parent class.

5. Which of the following is not true about method overloading?

a. It performs within a class.
b. It performs in child class.
c. Parameter must be different.
d. Parameters data type must be different.

5.13.3 Fill in the Blanks

1. ……. is used to initialise the object.
2. Methods are called overridden when they have ……. name and ……. parameters.
3. ……. keyword is used in java to refer current object.
4. Constructor has ……. return type.
5. Run time polymorphism can be achieved by using ……

5.13.4 Theoretical Questions

1. Explain encapsulation. How does it differ from abstraction?

c. Parameter must be different.
d. Parameters data type must be different.

5.13.3 Fill in the Blanks

1. ……. is used to initialise the object.
2. Methods are called overridden when they have ……. name and ……. parameters.
3. ……. keyword is used in java to refer current object.
4. Constructor has ……. return type.
5. Run time polymorphism can be achieved by using …….

5.13.4 Theoretical Questions

1. Explain encapsulation. How does it differ from abstraction?
2. What are the characteristics of encapsulation?
3. How does C++ support the properties of encapsulation and data hiding?
4. What is inheritance? Explain the types of inheritance.
5. What is virtual function and how it can be used in polymorphism?

Chapter 6 : Functional Programming

Learning Objectives

At the end of this module, you will be able to:

- Explain functional programming
- Discuss Haskell programming language
- Discuss types and typeclasses in Haskell
- Explain functions in Haskell
- Explain evaluation order
- Discuss high-order functions

6.1 Introduction

Functional programming languages are based on mathematical functions and logic programming languages are based on the predicate logic — Boolean logic. Lisp is

- Discuss Haskell programming language
- Discuss types and typeclasses in Haskell
- Explain functions in Haskell
- Explain evaluation order
- Discuss high-order functions

6.1 Introduction

Functional programming languages are based on mathematical functions and logic programming languages are based on the predicate logic — Boolean logic. Lisp is a programming language that combines functional programming with some imperative constructs.

Haskell is a widely used pure functional language. In imperative languages, such as C, the tasks are performed by assigning the computer its sequence and then it executes them. For example, you set a variable x to 2 and then perform some tasks and then set it to something else. In purely functional programming, we do not tell computer what to do. However, we explain everything in the form of functions.

In this chapter, you will learn about Haskell and its features, and how to set up an environment for Haskell. You will also learn about the data types and some basic operators used in Haskell; about types and typeclasses; and functions in Haskell as well. You will learn about monads and monoids in Haskell. And, at the end of this session, you will learn about high-order functions in Haskell.

SYBGEN ebooks Portal › Course
ebooks.sybgenlearning.com/appLms/index.php?modname=organization&op=custom_enditem&id_item=82

6.2 Introduction to Functional Programming

Functional programming is a framework that is used with a set of assumptions and depends on the particular ways of thinking through problems. It also uses specific methodologies to solve those problems.

The main difference between the functional programming paradigm and other programming paradigm is that the former use math functions instead of statements to express ideas. It means that a user writes math functions to solve a problem instead of specific functions. It also does not maintain the state. The state helps in tracking the values between function calls. Other paradigms use state to provide various results on the basis of environment. Functional programs are more predictable than those programs which support state. Since the functional programs do not maintain state, the data they work with is immutable. In order to change the value of the variable, you need to create a new variable.

6.2.1 Goals of Functional Programming

In imperative programming, the data moves through a series of steps in an order to produce a particular result. This process is fixed and rigid, and every time when a new result is needed, a new assembly line is required to be built for an application. In object-oriented programming (OOP), the steps are hidden but they remain same. It still does not allow to arrange the code in unexpected ways because of the underlying interdependencies of the code.

Waiting for ebooks.sybgenlearning.com...
Type here to search
ENG 22:26 22-11-2020

6.2.1 Goals of Functional Programming

In imperative programming, the data moves through a series of steps in an order to produce a particular result. This process is fixed and rigid, and every time when a new result is needed, a new assembly line is required to be built for an application. In object-oriented programming (OOP), the steps are hidden but they remain same. It still does not allow to arrange the code in unexpected ways because of the underlying interdependencies of the code.

Functional programming does not have any interdependencies because it replaces the procedures with the pure functions that use the immutable state. An application can change the data using the same methodologies used in pure math. The usage of the pure functions creates a flexible environment where the code order depends on the underlying math. The math helps in creating the real-world environment and it changes with the environment. In functional programming, it is easy to modify the code because the person implementing the change must understand only the math. A functional code can be created as long as a person understands the math model and its relationship to the real world.

Function programming also allows the unique coding approaches, such as the capability to pass a function to another function as input. This helps in changing the application behaviour in a predictable manner.

Programming languages that use the pure approach depend on the lambda calculus principles. Haskell, Lisp, Racket, Erlang, and OCaml are some of these programming languages.

Programming languages that use the impure approach adopt the benefit of functional programming but also not give up their own benefits. While working with an impure language, you need to take extra care because the code does not work in a purely functional manner. Python is one such programming language that supports multiple programming paradigms.

6.2.2 Using Functional Programming to Perform Tasks

The base of functional programming is lambda calculus which is a math abstraction. In addition to lambda functions, languages that use functional programming paradigm have some of the following common features:

6.2.1 Goals of Functional Programming

In imperative programming, the data moves through a series of steps in an order to produce a particular result. This process is fixed and rigid, and every time when a new result is needed, a new assembly line is required to be built for an application. In object-oriented programming (OOP), the steps are hidden but they remain same. It still does not allow to arrange the code in unexpected ways because of the underlying interdependencies of the code.

Functional programming does not have any interdependencies because it replaces the procedures with the pure functions that use the immutable state. An application can change the data using the same methodologies used in pure math. The usage of the pure functions creates a flexible environment where the code order depends on the underlying math. The math helps in creating the real-world environment and it changes with the environment. In functional programming, it is easy to modify the code because the person implementing the change must understand only the math. A functional code can be created as long as a person understands the math model and its relationship to the real world.

Function programming also allows the unique coding approaches, such as the capability to pass a function to another function as input. This helps in changing the application behaviour in a predictable manner.

Programming languages that use the pure approach depend on the lambda calculus principles. Haskell, Lisp, Racket, Erlang, and OCaml are some of these programming languages.

Programming languages that use the impure approach adopt the benefit of functional programming but also not give up their own benefits. While working with an impure language, you need to take extra care because the code does not work in a purely functional manner. Python is one such programming language that supports multiple programming paradigms.

6.2.2 Using Functional Programming to Perform Tasks

The base of functional programming is lambda calculus which is a math abstraction. In addition to lambda functions, languages that use functional programming paradigm have some of the following common features:

6.2.1 Goals of Functional Programming

In imperative programming, the data moves through a series of steps in an order to produce a particular result. This process is fixed and rigid, and every time when a new result is needed, a new assembly line is required to be built for an application. In object-oriented programming (OOP), the steps are hidden but they remain same. It still does not allow to arrange the code in unexpected ways because of the underlying interdependencies of the code.

Functional programming does not have any interdependencies because it replaces the procedures with the pure functions that use the immutable state. An application can change the data using the same methodologies used in pure math. The usage of the pure functions creates a flexible environment where the code order depends on the underlying math. The math helps in creating the real-world environment and it changes with the environment. In functional programming, it is easy to modify the code because the person implementing the change must understand only the math. A functional code can be created as long as a person understands the math model and its relationship to the real world.

Function programming also allows the unique coding approaches, such as the capability to pass a function to another function as input. This helps in changing the application behaviour in a predictable manner.

Programming languages that use the pure approach depend on the lambda calculus principles. Haskell, Lisp, Racket, Erlang, and OCaml are some of these programming languages.

Programming languages that use the impure approach adopt the benefit of functional programming but also not give up their own benefits. While working with an impure language, you need to take extra care because the code does not work in a purely functional manner. Python is one such programming language that supports multiple programming paradigms.

6.2.2 Using Functional Programming to Perform Tasks

The base of functional programming is lambda calculus which is a math abstraction. In addition to lambda functions, languages that use functional programming paradigm have some of the following common features:

6.2.1 Goals of Functional Programming

In imperative programming, the data moves through a series of steps in an order to produce a particular result. This process is fixed and rigid, and every time when a new result is needed, a new assembly line is required to be built for an application. In object-oriented programming (OOP), the steps are hidden but they remain same. It still does not allow to arrange the code in unexpected ways because of the underlying interdependencies of the code.

Functional programming does not have any interdependencies because it replaces the procedures with the pure functions that use the immutable state. An application can change the data using the same methodologies used in pure math. The usage of the pure functions creates a flexible environment where the code order depends on the underlying math. The math helps in creating the real-world environment and it changes with the environment. In functional programming, it is easy to modify the code because the person implementing the change must understand only the math. A functional code can be created as long as a person understands the math model and its relationship to the real world.

Function programming also allows the unique coding approaches, such as the capability to pass a function to another function as input. This helps in changing the application behaviour in a predictable manner.

Programming languages that use the pure approach depend on the lambda calculus principles. Haskell, Lisp, Racket, Erlang, and OCaml are some of these programming languages.

Programming languages that use the impure approach adopt the benefit of functional programming but also not give up their own benefits. While working with an impure language, you need to take extra care because the code does not work in a purely functional manner. Python is one such programming language that supports

SYBGEN ebooks Portal › Course
ebooks.sybgenlearning.com/appLms/index.php?modname=organization&op=custom_enditem&id_item=82

6.2.2 Using Functional Programming to Perform Tasks

The base of functional programming is lambda calculus which is a math abstraction. In addition to lambda functions, languages that use functional programming paradigm have some of the following common features:

- **First-class and higher-order functions:** These functions allow you to provide a function as an input.
- **Pure functions:** A pure function has no side effect. It can be removed if no function depends on its output. You can obtain the same output if the function is called with a given set of outputs. The function calls can be processed parallelly without the consequences.
- **Recursion:** In the implementations of function language, recursion is used to implement looping. Recursion works in a different manner in functional languages because there is no change in the application state.
- **Referential transparency:** The variable value does not change in a functional language because there is no assignment operator in function languages.

There are various considerations to perform the task in the functional programming languages. However, these issues are not consistent. For example, some languages use strict evaluation while other languages use non-strict evaluation. In strict evaluation, the language first checks the function completely before evaluating it. In non-strict evaluation, the function fails if the failing term is used to create an output. The Miranda, Clean, and Haskell languages all implement non-strict evaluation.

Various functional languages implement various type systems, so the manner in which the underlying computer detects the type of a value changes from language to language. These languages also have their own data structures.

Screenshot saved
The screenshot was added to your OneDrive.
OneDrive

Type here to search
ENG 22:27 22-11-2020

6.3 Getting Started with Haskell

Haskell is a purely functional programming language. In functional programming, computer programs are treated as a mathematical function. When functions are programmed in purely functional style, we do not focus on manipulating states and variables but on constants and functions. A functional program is a single expression, which is executed by evaluating the expression. Some other functional programming languages include Lisp, Python, Erlang, Racket, F#, Clojure, etc.

It is tough and expensive to write large software systems and it is equivalently difficult and expensive to maintain those systems. Functional programming languages, such as Haskell, can make it easier and cheaper. Haskell has a shorter, clearer and more maintainable code. There is a smaller "semantic gap" between the programmer and the language. It is suitable for programs that need to be modified and maintained regularly.

6.3.1 Features of Haskell

Some of the common features of Haskell are:

- **Statically typed**: In conventional programming language, we need to define variables with their types. In contrast, Haskell is considered as a statically typed language. It means that when we will compile our program, the compiler will know which piece of code is a string, which is a number, etc. It can identify which type of variable is declared.

6.3.1 Features of Haskell

Some of the common features of Haskell are:

- **Statically typed**: In conventional programming language, we need to define variables with their types. In contrast, Haskell is considered as a statically typed language. It means that when we will compile our program, the compiler will know which piece of code is a string, which is a number, etc. It can identify which type of variable is declared.
- **Purely functional**: Haskell is a purely functional language. Every function in a Haskell is a purely mathematical function. A function when invoked using the same arguments will return the same result every time.
- **Type inference**: You do not need to explicitly define every type being used in a Haskell program. Type can be inferred by unifying every type bidirectionally. Suppose, if we will type x = 8 – 4, then we will not need to specify that x is a number. It allows the code to be more general. If a function takes two parameters and adds them together and there is no type defined, then the function will work on any two parameters that act like numbers.
- **Concurrent**: Haskell allows concurrent programming due to its explicit handling of effects. Its compiler has a light-weight concurrency library containing a number of useful concurrency primitives.
- **Lazy**: Haskell is a lazy language. By lazy, it is meant that Haskell does not evaluate any express without a reason. Functions only evaluate their arguments when required.
- **Packages**: Haskell has a wide range of packages that are available on the public package servers.

6.3.2 Environment Setting for Haskell

In order to run Haskell programs, we need a special program known as a Haskell compiler. It takes the code written in Haskell and then translates it into machine code.

There are two commonly known compilers which can be used with Haskell — Hugs and GHC. Hugs is an interpreter which means that it cannot be used for compiling stand-alone programs. It is used for testing and debugging programs in an interactive environment.

Glasgow Haskell Compiler (GHC) is an interpreter and a compiler whose interactive environment is slower than Hugs. It allows function definitions in the environment and implements Haskell 98 and extensions.

You can go to https://www.haskell.org/downloads/ for the latest instructions to download the complier. In Windows, you need to first install the Chocolatey, which is the software management solution. Let us download the Chocolatey and then Haskell platform installer using the following steps:

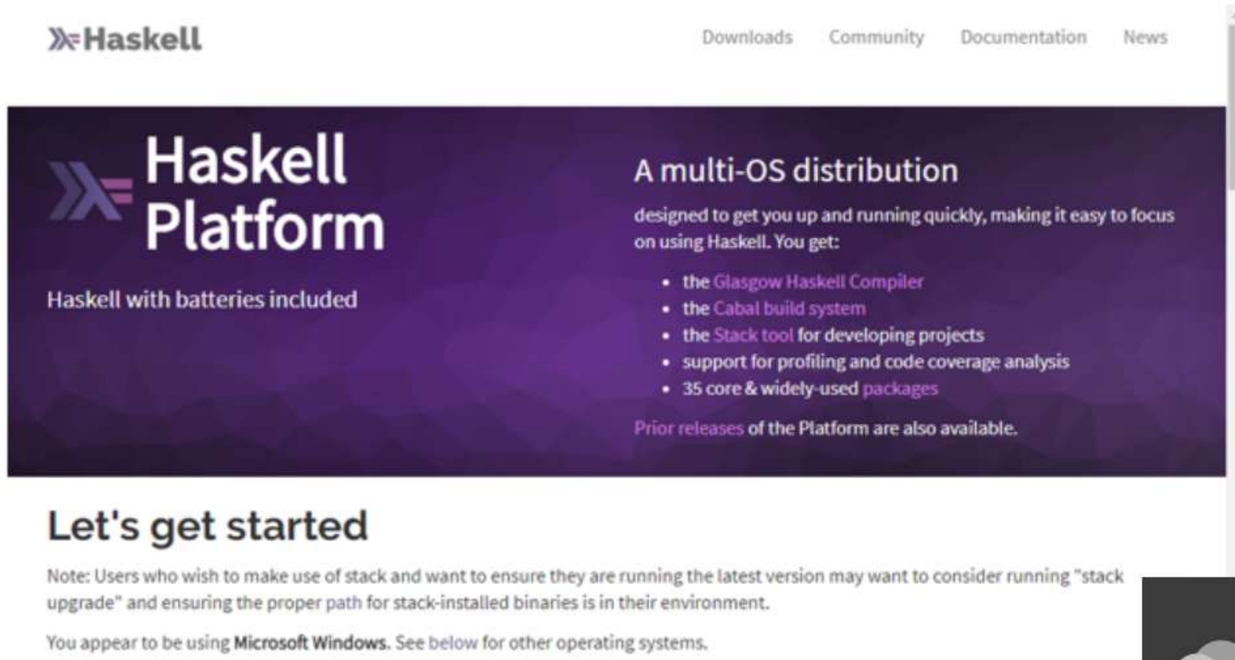
1. Open a web browser and go to the website https://www.haskell.org/platform/windows.html. Figure 1 shows the Haskell Platform installer website:

SYBGEN ebooks Portal › Course
ebooks.sybgenlearning.com/appLms/index.php?modname=organization&op=custom_enditem&id_item=82

6.3 Getting Started with Haskell

You can go to https://www.haskell.org/downloads/ for the latest instructions to download the complier. In Windows, you need to first install the Chocolatey, which is the software management solution. Let us download the Chocolatey and then Haskell platform installer using the following steps:

1. Open a web browser and go to the website https://www.haskell.org/platform/windows.html. Figure 1 shows the Haskell Platform installer website:



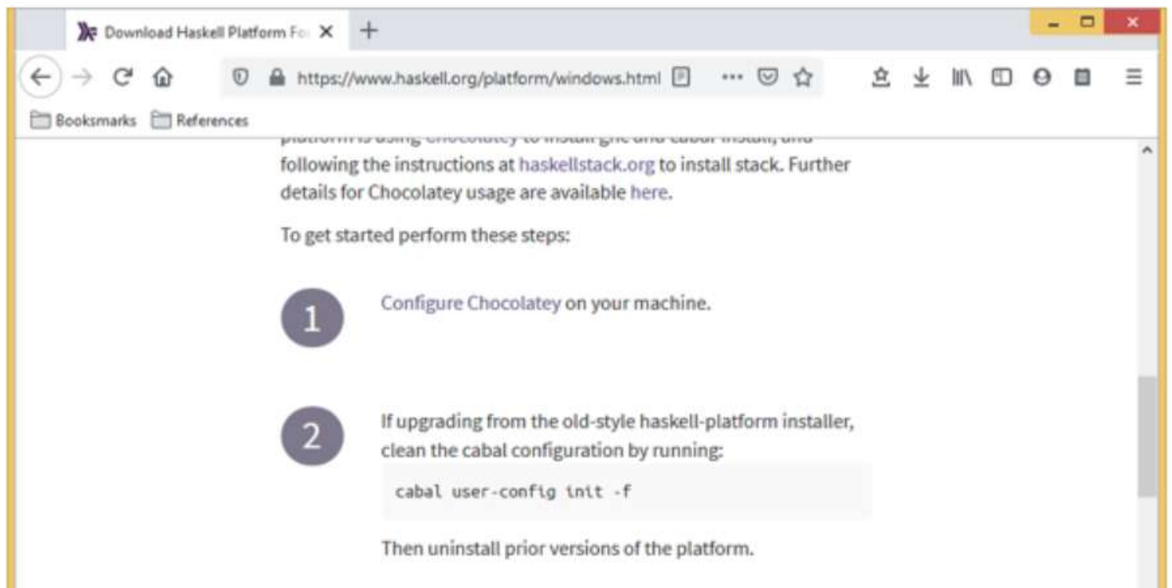
Screenshot saved
The screenshot was added to your OneDrive.
OneDrive
Type here to search
ENG
22:28
22-11-2020

6.3 Getting Started with Haskell

You appear to be using **Microsoft Windows.** See below for other operating systems.

Figure 1: Haskell Platform Installer Website

2, Scroll down the website and click the Configure Chocolatey link to open the configuration page of Chocolatey, as shown in Figure 2:



SYBGEN ebooks Portal › Course
ebooks.sybgenlearning.com/appLms/index.php?modname=organization&op=custom_enditem&id_item=82

details for Chocolatey usage are available here.

To get started perform these steps:

1. Configure Chocolatey on your machine.

2. If upgrading from the old-style haskell-platform installer, clean the cabal configuration by running:

```
cabal user-config init -f
```

Then uninstall prior versions of the platform.

3. At an elevated command prompt, run:

```
choco install haskell-dev
refreshenv
```

Figure 2: Configure Chocolatey Link

3. Open the PowerShell with the administrative privileges

4. Type `Get-ExecutionPolicy` and run it. If it returns Restricted then run `Set-ExecutionPolicy AllSigned` or `Set-Exe`
as shown in Figure 3:

Screenshot saved
The screenshot was added to your OneDrive.
OneDrive

Type here to search
ENG
22:28
22-11-2020

6.3 Getting Started with Haskell

3. Open the PowerShell with the administrative privileges

4. Type `Get-ExecutionPolicy` and run it. If it returns Restricted then run `Set-ExecutionPolicy AllSigned` or `Set-ExecutionPolicy Bypass -Scope Process`, as shown in Figure 3:

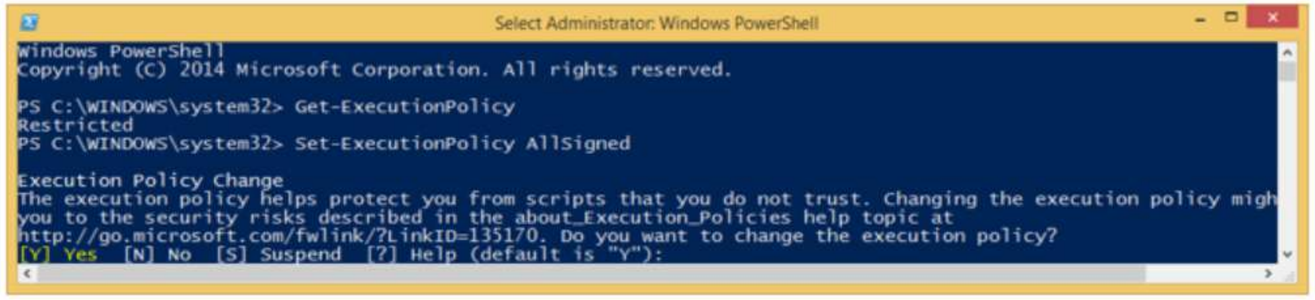


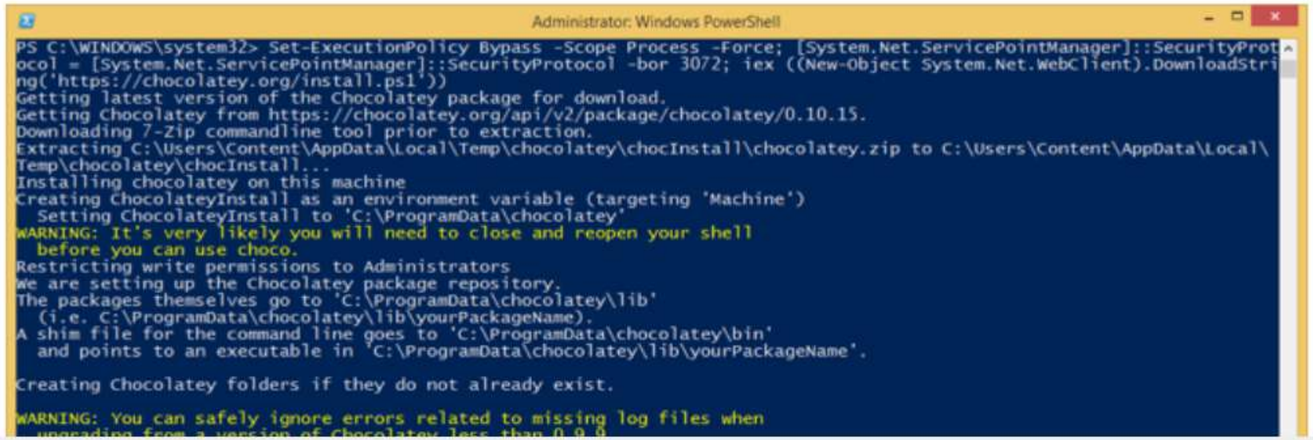
Figure 3: Changing Execution Policy

5. Run the following command as shown in Figure 4:

```
Set-ExecutionPolicy    Bypass    -Scope    Process    -Force;
[System.Net.ServicePointManager]::SecurityProtocol          =
[System.Net.ServicePointManager]::SecurityProtocol  -bor  3072;
iex                                                  ((New-Object
```

5. Run the following command as shown in Figure 4:

```
Set-ExecutionPolicy Bypass -Scope Process -Force;
[System.Net.ServicePointManager]::SecurityProtocol =
[System.Net.ServicePointManager]::SecurityProtocol -bor 3072;
iex ((New-Object
System.Net.WebClient).DownloadString('https://chocolatey.org/i
nstall.ps1'))
```



6.3 Getting Started with Haskell

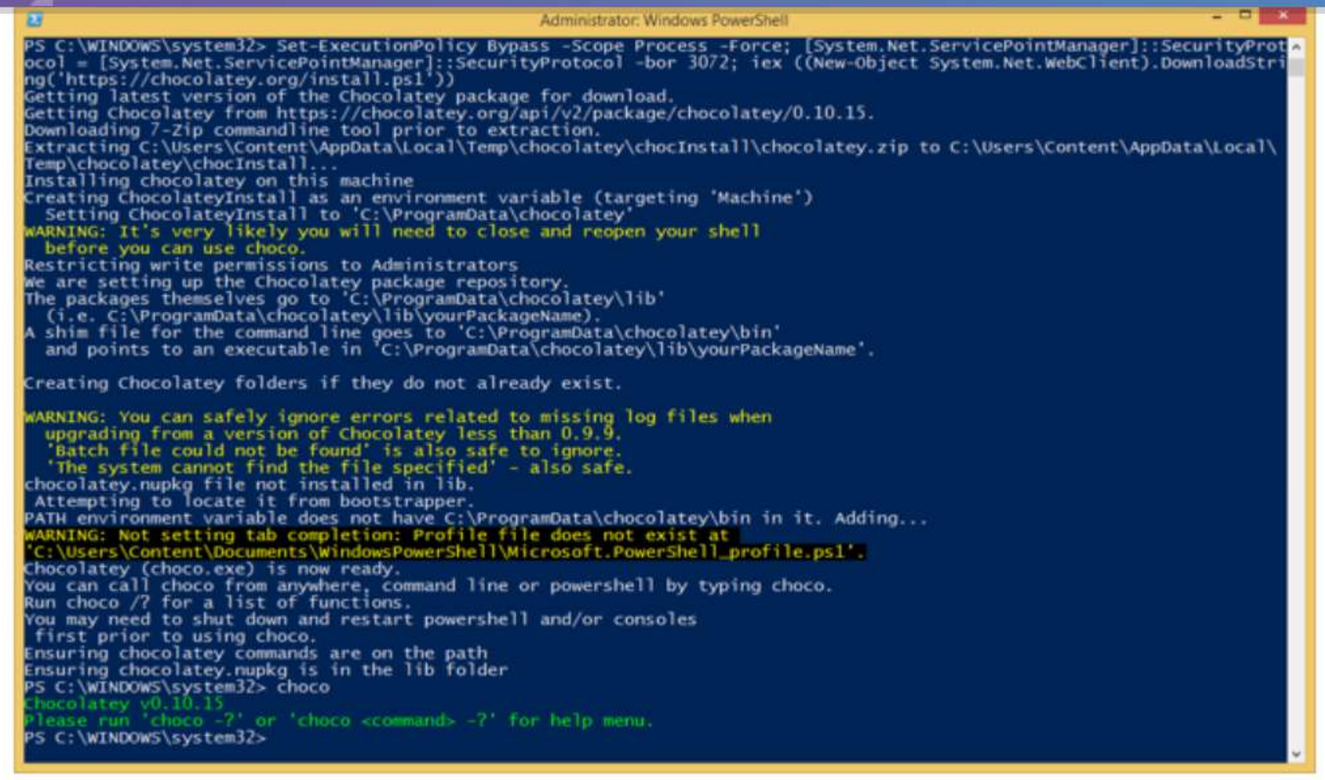


Figure 4: Setting the Execution Policy

6. Type choco or choco -? to check if it is installed properly, as shown in Figure 5:

Type here to search
ENG
22:28
22-11-2020

6. Type choco or choco -? to check if it is installed properly, as shown in Figure 5:

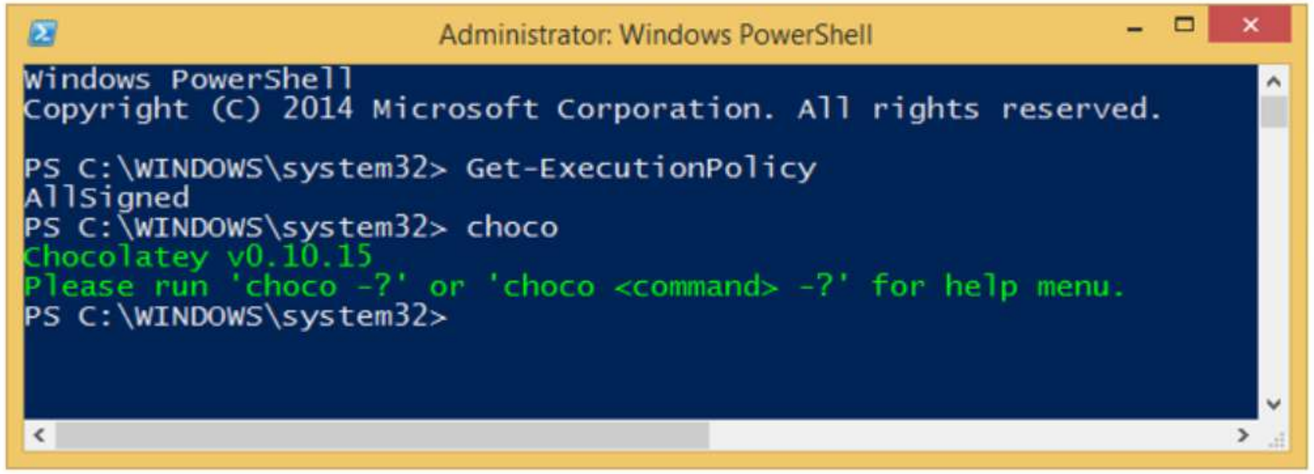


Figure 5: Checking Chocolatey

7.Run choco install haskell-dev to install Haskell, as shown in Figure 6:

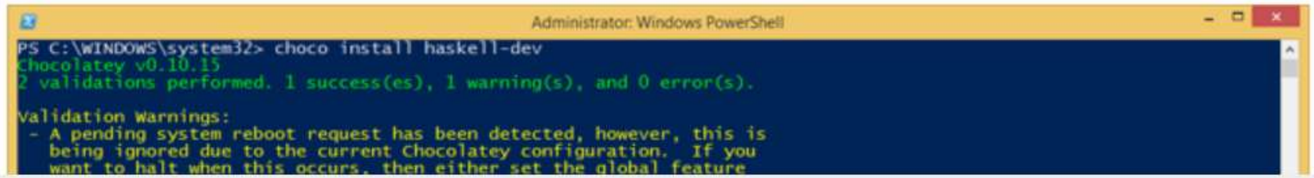


Figure 5: Checking Chocolatey

7.Run choco install haskell-dev to install Haskell, as shown in Figure 6:

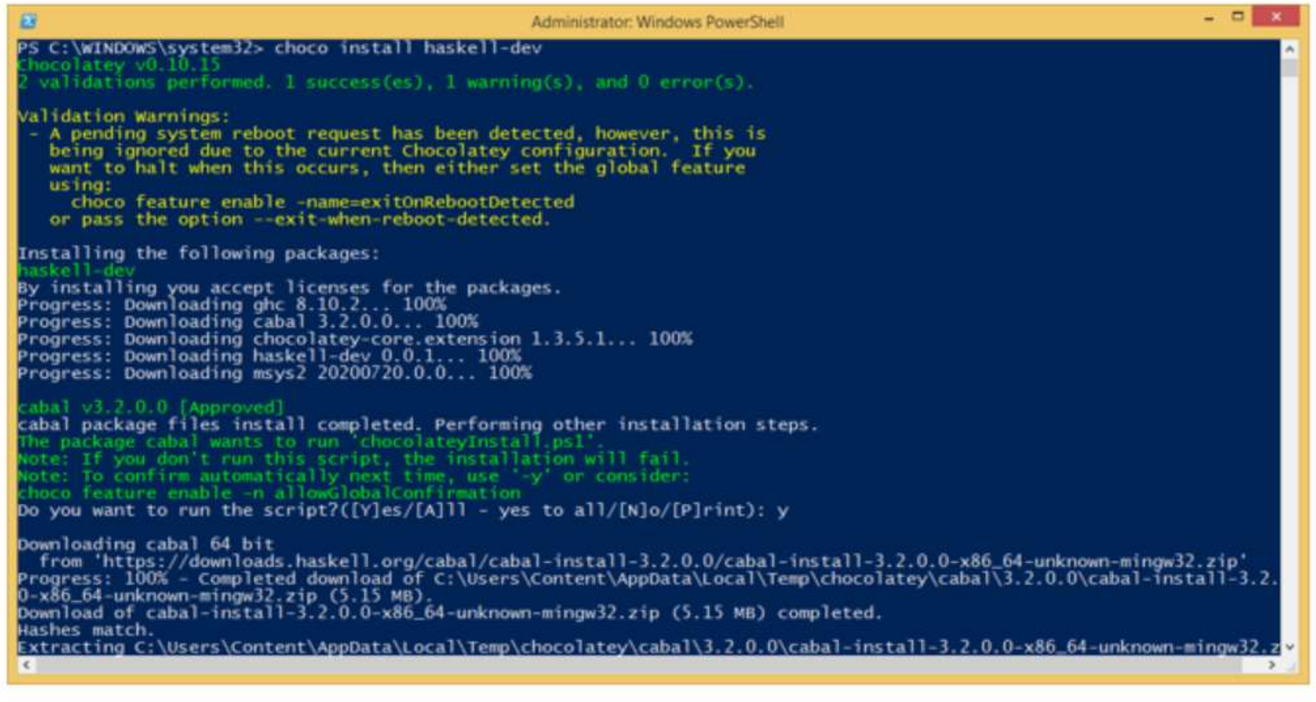


Figure 6: Installing Haskell

Figure 6: Installing Haskell

After the complete installation, run ghc --help command to check if it properly installed, as shown in Figure 7

Administrator: Windows PowerShell

```
PS C:\WINDOWS\system32> ghc --help
Usage:

    ghc.exe [command-line-options-and-input-files]

To compile and link a complete Haskell program, run the compiler like
so:

    ghc.exe Main

where the module Main is in a file named Main.hs (or Main.lhs) in the
current directory.  The other modules in the program will be located
and compiled automatically, and the linked program will be placed in
the file `Main' (or `Main.exe' on Windows).

Alternatively, ghc.exe can be used to compile files individually.  Each
input file is guided through (some of the) possible phases of a
compilation:

    - unlit:    extract code from a "literate program"
    - hscpp:    run code through the C pre-processor (if -cpp flag given)
    - hsc:      run the Haskell compiler proper
    - gcc:      run the C compiler (if compiling via C)
    - as:       run the assembler
    - ld:       run the linker

For each input file, the phase to START with is determined by the
file's suffix:
```

```
input file is guided through (some of the) possible phases of a
compilation:

    - unlit:    extract code from a "literate program"
    - hscpp:    run code through the C pre-processor (if -cpp flag given)
    - hsc:      run the Haskell compiler proper
    - gcc:      run the C compiler (if compiling via C)
    - as:       run the assembler
    - ld:       run the linker

For each input file, the phase to START with is determined by the
file's suffix:

    - .lhs      literate Haskell                unlit
    - .hs       plain Haskell                   ghc
    - .hc       C from the Haskell compiler     gcc
    - .c        C not from the Haskell compiler gcc
    - .s        assembly language               as
```

Figure 7: Checking GHC Installation

6.3.3 Compiling Haskell

It is possible to perform the tasks in the interpreter, but you can also load modules and interpret them. You load it into the interpreter and then execute it. Let us create a text file and name it as hello.hs. Type the following code into the file and save it on disk:

main = putStrLn "Hello, it is fun learning Haskell!"

This code just prints a line. After saving the file, you can use GHCi to run the application. The following steps are used to load, test and compile the application:

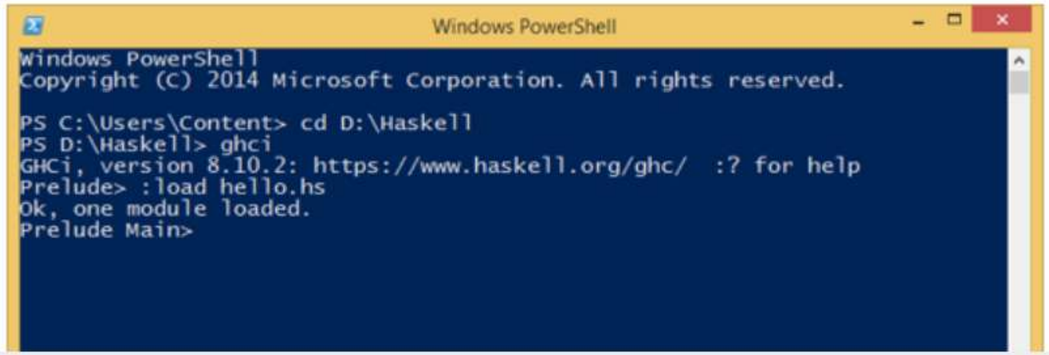
6.3.3 Compiling Haskell

It is possible to perform the tasks in the interpreter, but you can also load modules and interpret them. You load it into the interpreter and then execute it. Let us create a text file and name it as hello.hs. Type the following code into the file and save it on disk:

```
main = putStrLn "Hello, it is fun learning Haskell!"
```

This code just prints a line. After saving the file, you can use GHCi to run the application. The following steps are used to load, test and compile the application:

1. Type :cd <Source Code Directory> and press **Enter**.
2. Type :load hello.hs and press Enter, as shown in Figure :



6.3 Getting Started with Haskell

1. Type :cd <Source Code Directory> and press **Enter**.
2. Type :load hello.hs and press Enter, as shown in Figure :

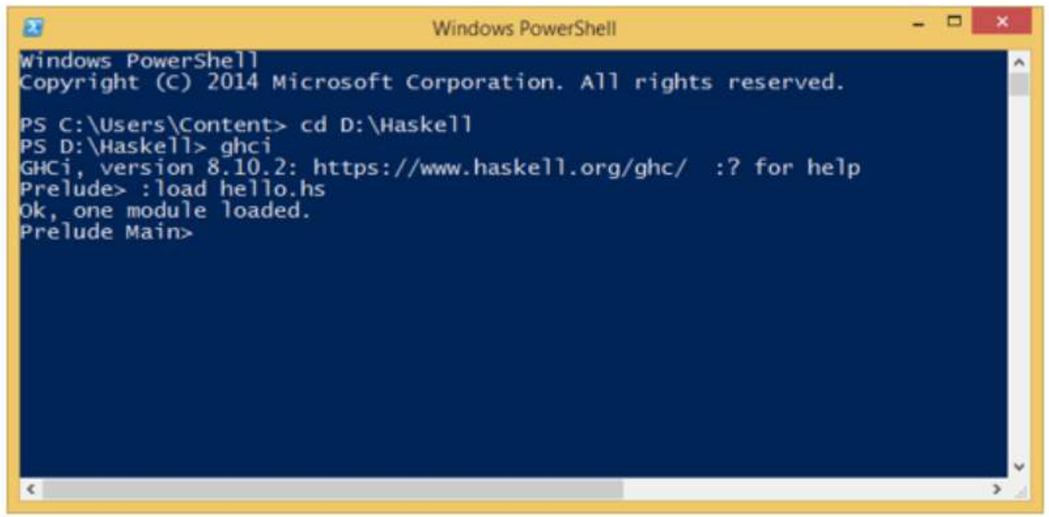
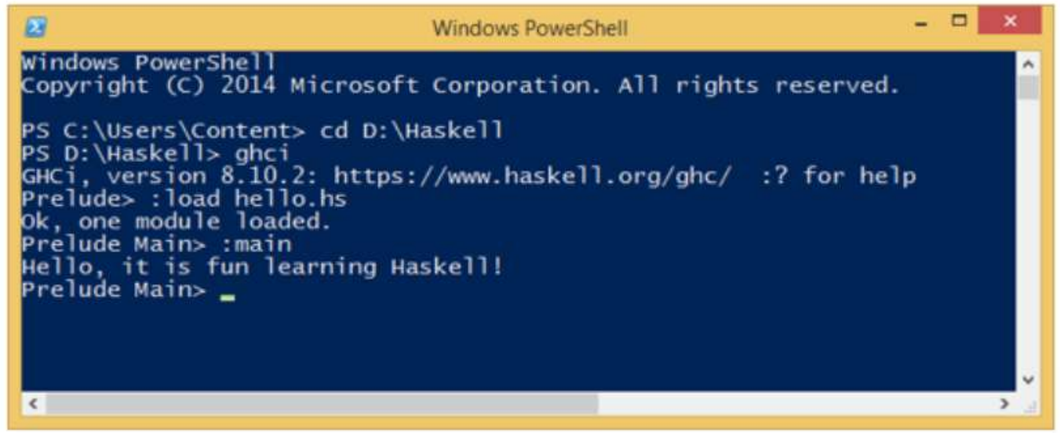


Figure 8: Loading the Application

3. Type `:main` and press Enter, as shown in Figure 9:

Figure 8: Loading the Application

3. Type `:main` and press Enter, as shown in Figure 9:



4. Type :quit and press **Enter**.

The interpreter closes.

6.3.4 Data Types in Haskell

Haskell is a pure functional programming language. It is more interactive and intelligent than other conventional programming languages. In this section, we will discuss some of the basic data models that are predefined.

- **Numbers:** Haskell is able to decode any kind of number as a number. It is not necessary to mention its type. Simple arithmetic performed between two numbers works similar way to the languages, such as C and Python. The expressions are written in infix form, where an operator appears between its operands. Now, let's go to the Prelude command prompt and run 4+6. Figure 10 shows its output:

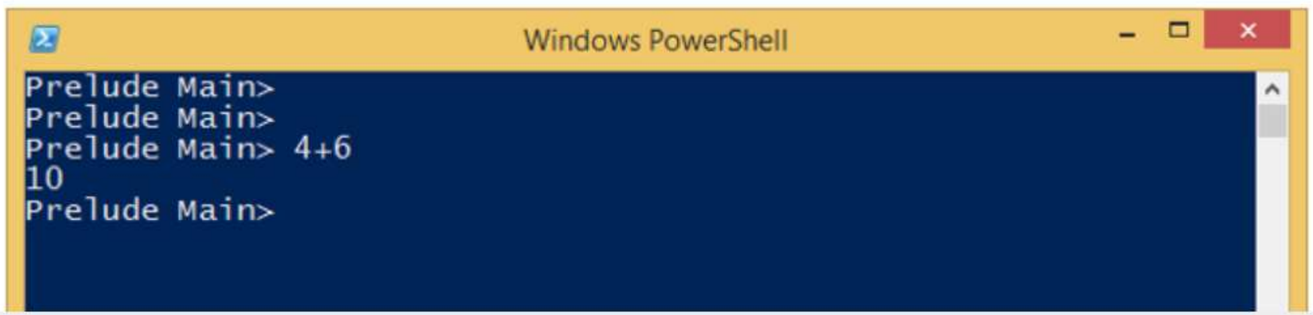


Figure 10: Simple Addition of Numbers

As shown in Figure 10, two numbers are passed as arguments to the GHCI compiler without predefining their type. However, the compiler decoded them as numbers. Let's try another mathematical calculation, such as 20+(2*8)-15. Figure 11 shows its output:

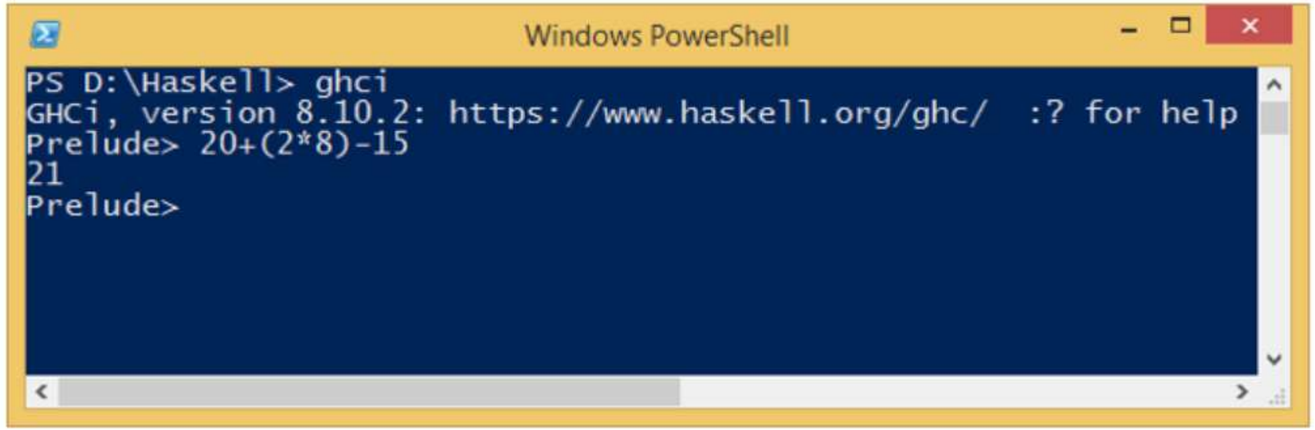


Figure 11: Simple Mathematical Expression

Figure 11: Simple Mathematical Expression

We can also write an expression in the prefix form, where the operator precedes its arguments. In prefix form, we need to enclose the operator in parentheses as shown in Figure 12:

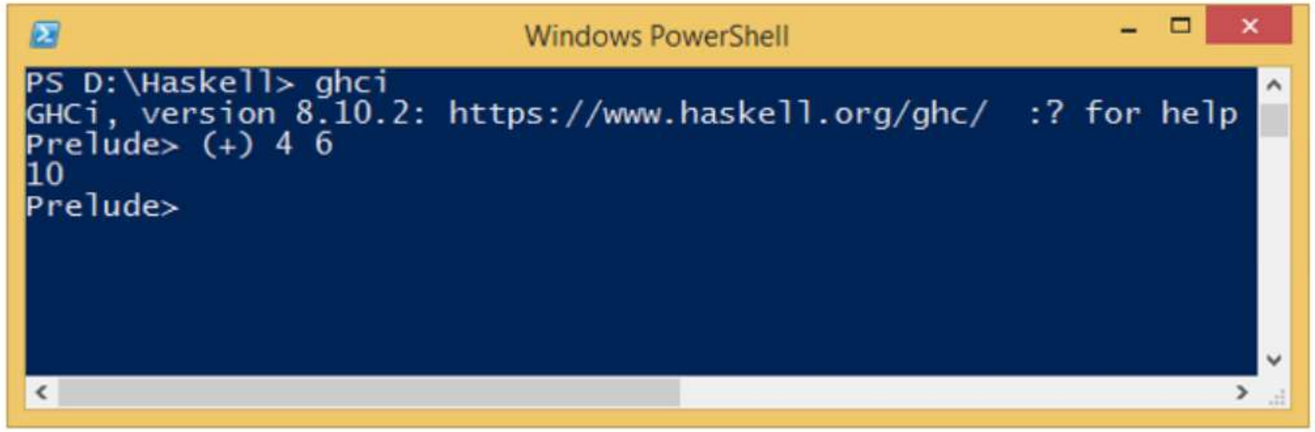


Figure 12: Mathematical Expression in Prefix Form

- **Characters:** Haskell can also identify a character given in as an input to it. A single character is enclosed in single quotes. Let's provide the following line as input and check its output as shown in Figure 13:

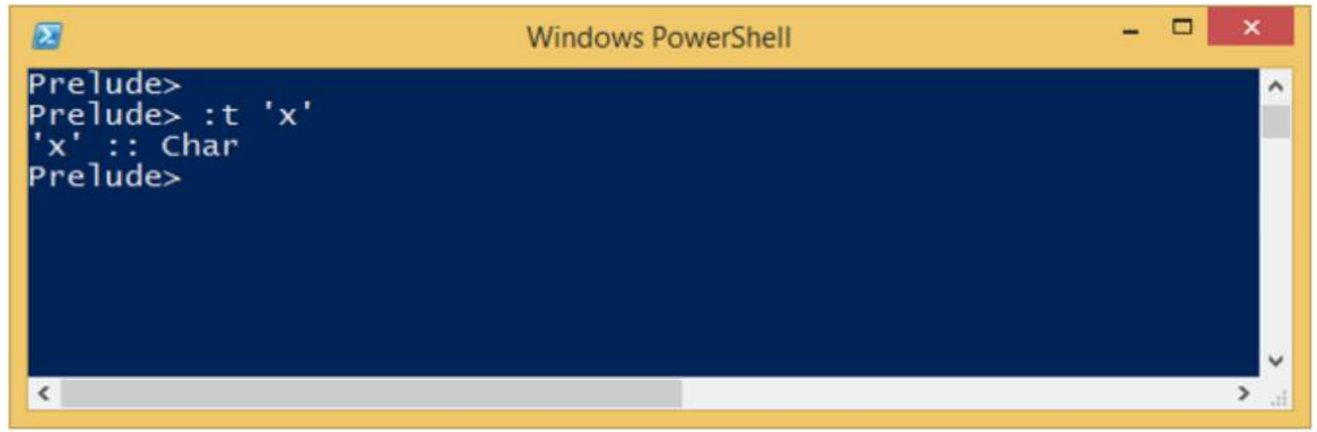


Figure 13: Checking Type of Input

The :t option is used to supply the input. As shown in Figure 13, :t is used to include the specific type related to the inputs.

- **Strings:** A string is a collection of characters. In Haskell, a string is notated using double quotes as shown in Figure 14:

6.3 Getting Started with Haskell

- **Strings:** A string is a collection of characters. In Haskell, a string is notated using double quotes as shown in Figure 14:

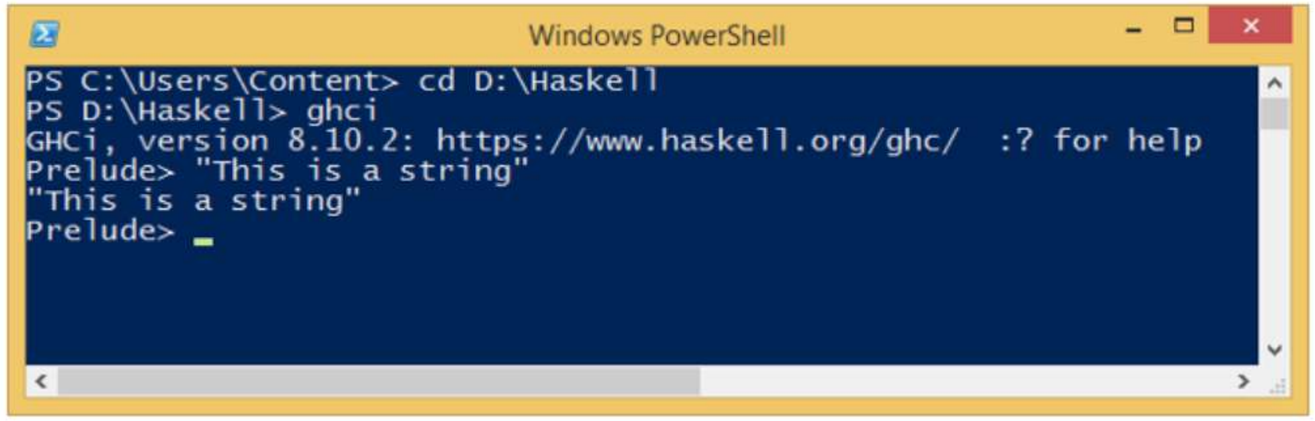


Figure 14: Example of String

In Haskell, hard-to-see characters are represented by “escaping” them. The escape characters and their rules are similar to the widely used conventions established by the C language. For example, \n denotes a newline and \t denotes a tab. Figure 15 shows an example:

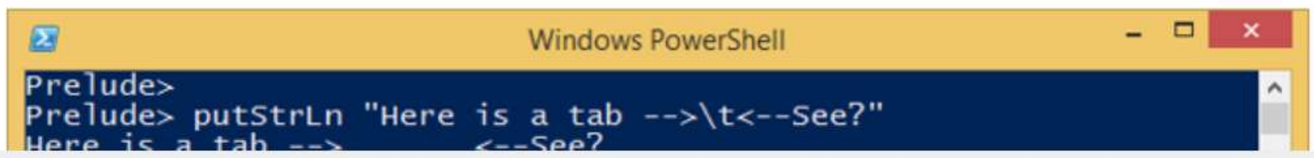
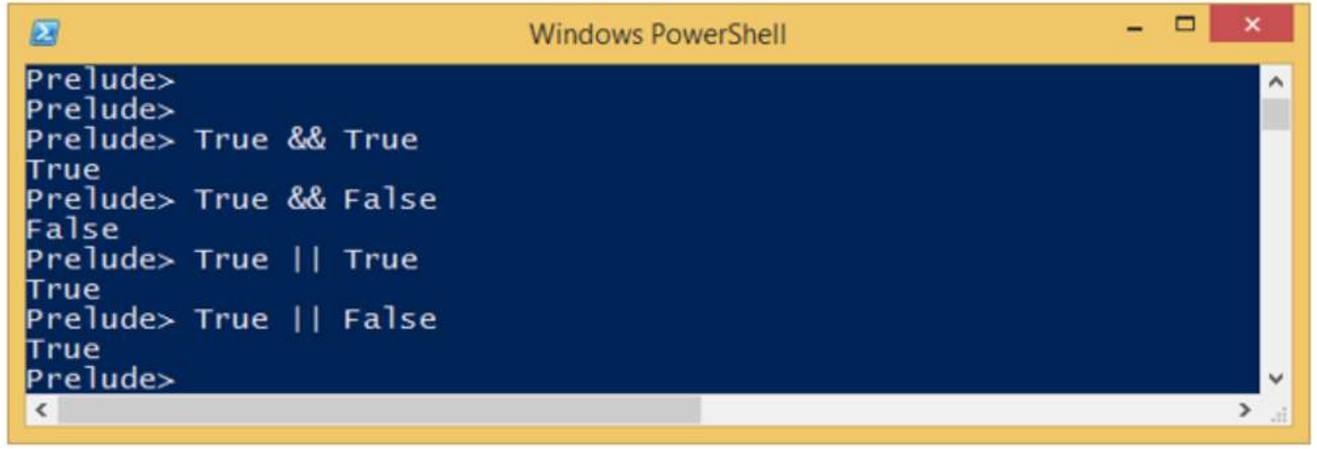


Figure 15: Examples of Escape Characters

As shown in Figure 15, the putStrLn function prints a string.

- **Boolean:** The values of Boolean logic in Haskell are True and False. Haskell uses C-influenced operators for working with Boolean values that are (&&) is logical "and", and (||) is logical "or". Figure 16 shows some Boolean operations:



```
True
Prelude> True || False
True
Prelude>
```

Figure 16: Example of Boolean Inputs

- **Lists:** A list is a collection of same data types separated by comma. It is surrounded by square brackets as shown in Figure 17:

```
Windows PowerShell
Prelude>
Prelude> [1,2,3]
[1,2,3]
Prelude>
```

Figure 17: Example of a List

It can be of any length. An empty list is written as [] as shown in Figure 18:

It can be of any length. An empty list is written as [] as shown in Figure 18:

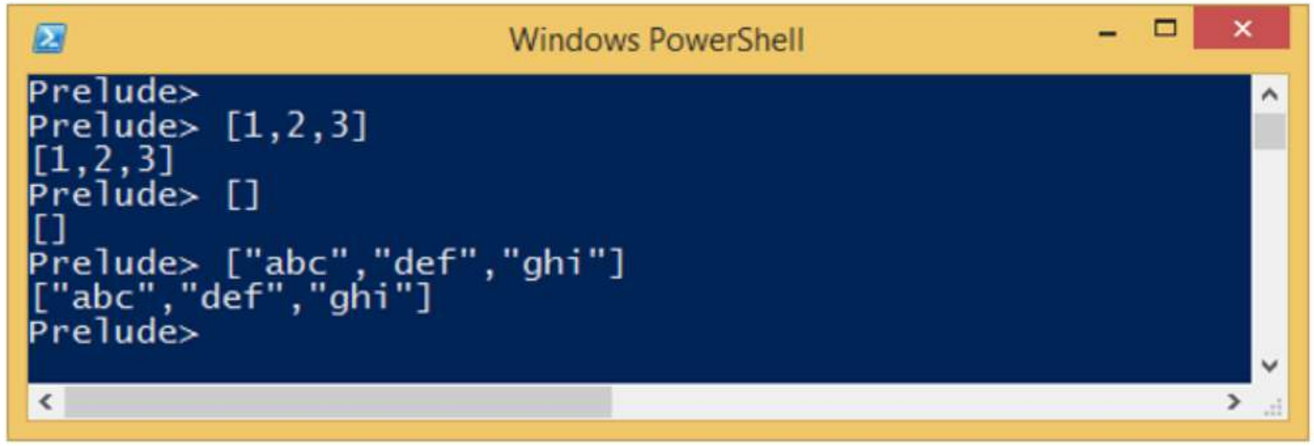
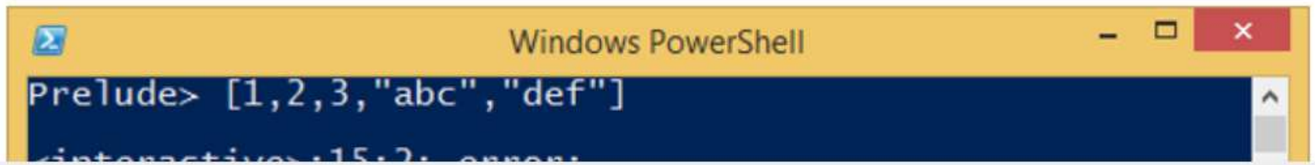


Figure 18: Example of an Empty List

A list should contain elements of the same type. The list shown in Figure 19 violates this rule and displays the error:



```
    * No instance for (Num [Char]) arising from the liter
    * In the expression: 1
      In the expression: [1, 2, 3, "abc", ....]
      In an equation for `it': it = [1, 2, 3, ....]
Prelude>
```

Figure 19: An Error Example

As shown in Figure 19, Haskell displays an error showing "No instance for (Num [Char]) arising from the literal `1'".

- **Tuple:** A tuple is a fixed-size collection of values, where each value can have a different type. This is different from a list which can have any length but must have elements of same type. Another difference is that a list is a homogeneous data type but a tuple is heterogeneous in nature.

  A tuple is represented by single parenthesis. Its elements are separated by commas. Suppose, we want to keep track of two pieces of information about a person: his/her name and age. We cannot keep both pieces of information in a list because they have

but a tuple is heterogeneous in nature.

A tuple is represented by single parenthesis. Its elements are separated by commas. Suppose, we want to keep track of two pieces of information about a person: his/her name and age. We cannot keep both pieces of information in a list because they have different types. Therefore, we will use a tuple as shown in Figure 20:

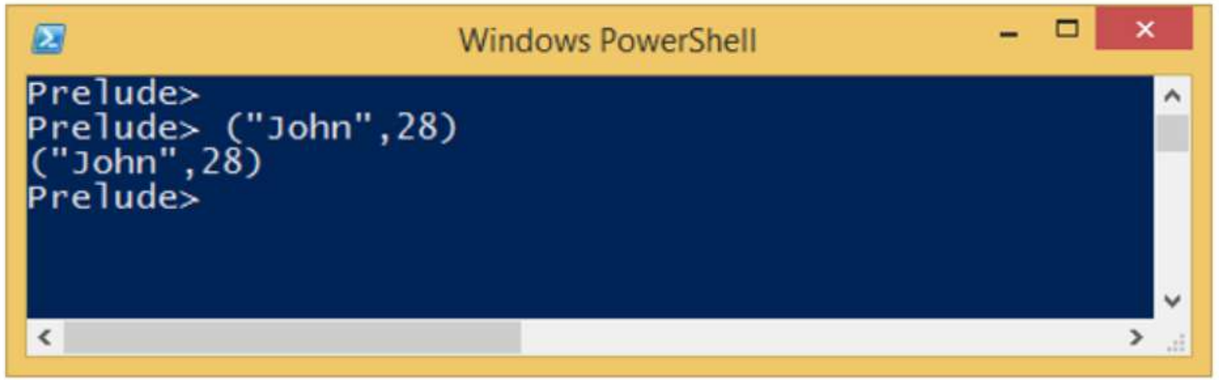


Figure 20: An Example of Tuple

A tuple of zero elements is denoted as (). Haskell does not support single element tuple. A 2-tuple has two elements and is usually called a pair. A 3-tuple (sometimes called a triple) has three elements; a 5-tuple has five; and so on.

A tuple of zero elements is denoted as (). Haskell does not support single element tuple. A 2-tuple has two elements and is usually called a pair. A 3-tuple (sometimes called a triple) has three elements; a 5-tuple has five; and so on.

6.3.5 Basic Operators of Haskell

Operators are the foundational blocks of a program. They define the functionality of the program. These fundamentals enable us to decide the flow of execution and the conditions to reroute the flow in a way.

An operator is the main element that manipulates or operates on values to produce the result. Operators are the special symbols, which carry out the arithmetic or logical computation on operands and generate an output. Operands are the values on which the operator operates.

- **Arithmetic Operators:** In Haskell, the mathematical calculations and operations, such as addition, subtraction, multiplication, division, remainder, exponential, etc. are carried out by arithmetic operators.

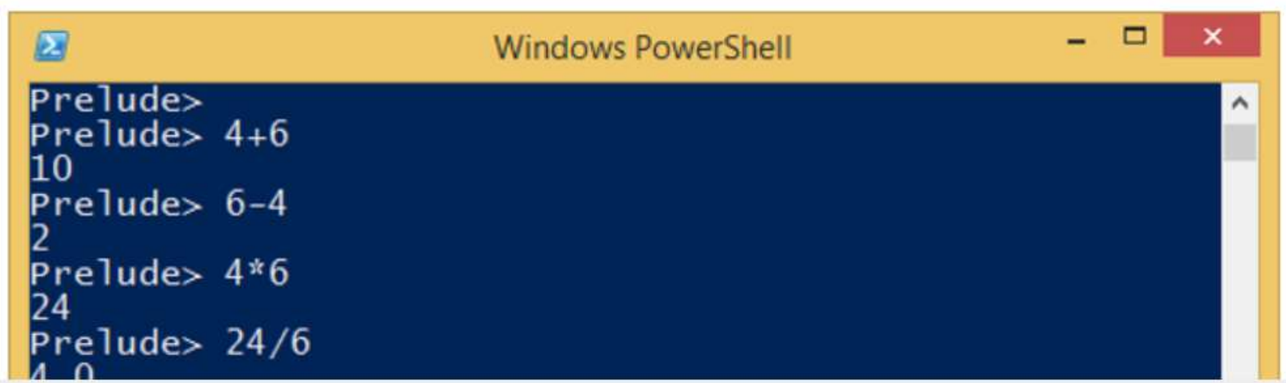
Table 1 gives the description of arithmetic operators and their operations:

Table 1: List of Arithmetic Operators

Operator	Description	Example
+	Addition of numbers and concatenation of data types	a + b

Operator	Description	Example
+	Addition of numbers and concatenation of data types	a + b
-	Subtraction of numbers and differences of sets	a - b
*	Multiplication of numbers and repetition of data types	a * b
/	Division of numbers	a / b

Figure 21 represents the code for implementation of different arithmetic operators:



```
Prelude> 4*6
24
Prelude> 24/6
4.0
Prelude>
```

Figure 21: Implementation of Different Arithmetic Operators

- **Relational Operators:** Values are compared to each other using relational operators in Haskell. The comparison produces the result as true or false. Table 2 gives the description of relational operators and their operations:

Table 2: List of Relational Operators

Operator	Description	Example
>	Value greater than	a > b
<	Value Less than	a < b
==	Value equals	a == b

SYBGEN ebooks Portal › Course
ebooks.sybgenlearning.com/appLms/index.php?modname=organization&op=custom_enditem&id_item=82

6.3 Getting Started with Haskell

Operator	Description	Example
>	Value greater than	a > b
<	Value Less than	a < b
==	Value equals	a == b
/=	Value is not equal	a /= b
>=	Value is greater than or equal	a >= b
<=	Value is less than or equal	a <= b

The implementation of these relational operators is shown in Figure 22:

Windows PowerShell

```
Prelude>
Prelude> 2 > 1
True
Prelude> 1 < 2
```


Type here to search
ENG
22:31
22-11-2020

Windows PowerShell

```
Prelude>
Prelude> 2 > 1
True
Prelude> 1 < 2
True
Prelude> 1 == 1
True
Prelude> 2 /= 1
True
Prelude> 2 >= 1.99
True
Prelude> 1 <= 1.01
True
Prelude>
```

Figure 22: Implementation of Different Relational Operators

6.3.6 Type and Type Classes

In Haskell, every expression and function have a type. For example, the True value has the Bool type, while the “abc” value has the String type. The type of a value shows that it contains some properties similar to the other values of the same type. You can say that an expression has type X, or is of type X.

6.3.6 Type and Type Classes

In Haskell, every expression and function have a type. For example, the True value has the Bool type, while the “abc” value has the String type. The type of a value shows that it contains some properties similar to the other values of the same type. You can say that an expression has type X, or is of type X.

A type system in Haskell provides an abstraction at the lowest level. It helps us to prevent accidently mixing those types. The benefit of low-level abstraction helps in remembering low-level details. For example, if there is a number value in a program, then there is no need to worry about how the value will behave.

Haskell’s type system helps in writing concise, powerful programs. The types in Haskell have three properties:

- They are strong
- They are static
- They can be automatically inferred

Let us discuss these properties:

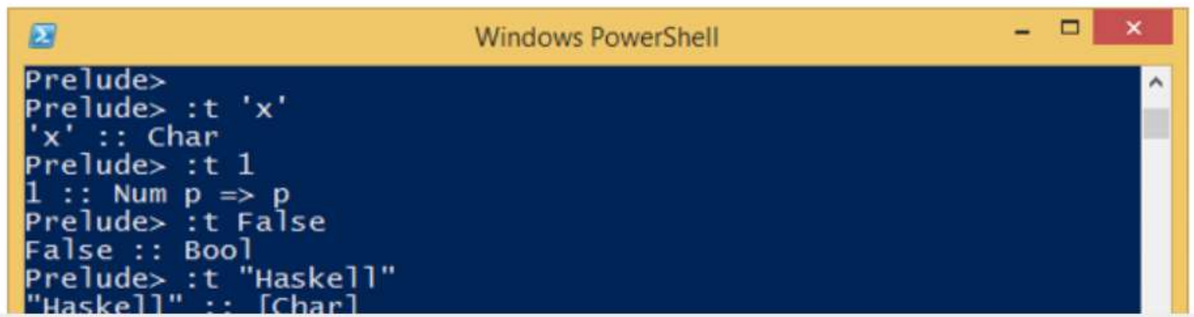
- **Strong Types:** A strong type system guarantees that a program cannot contain certain kinds of errors. These errors may occur due to writing expressions that do not make such sense. For example, a function, which is supposed to work with string, is being passed with the integers value. The Haskell compiler will reject this expression and cause a type error.

not make such sense. For example, a function, which is supposed to work with string, is being passed with the integers value. The Haskell compiler will reject this expression and cause a type error.

Haskell compiler also does not automatically force value to change from one type to another. For example, a C compile will automatically change an int value into a float if a function expects an int value. However, Haskell compiler will raise an error in such situation.

- **Static Types:** A static type system means that the compiler knows the type of every value and expression. However, Haskell supports programming with truly dynamic types. Haskell provides a combination of strong and static typing. This helps in avoiding type errors at runtime.
- **Type Inference:** A Haskell compile can automatically find the type of almost all expressions in a program. This is known as type inference.

Now, let's discuss how to use GHCI to find the types of some expressions. It can be done by the :type or :t command follows by any valid expressions. Figure 23 shows some examples:



```
    * Variable not in scope: (\=) :: t0 -> t1 -> t
    * Perhaps you meant one of these:
        `>=' (imported from Prelude), `==' (imported from Prelu
        `/=' (imported from Prelude)
Prelude>
```

Figure 23: Some Examples of Types

As you can see in Figure 23, the output is usually followed by :: and its type. :: is read as 'has type of'. The 'x' has a type of Char which stands for character. However, "Haskell" also has a type of [Char]. The square brackets denote a list. It means that it is a list of characters.

Functions also have types. They can be given an explicit type declaration at the time of writing. Suppose makeUppercase is a function that converts a string to a string of uppercase letters. This function will map from a string to a string and it will take one string as a parameter and returns another as a result. Its type can be declared in the following manner:

makeUppercase :: [Char] -> [Char]

If you want to declare that type of a function that takes several parameter, then you can declare in the following manner. Suppose subTwo is a function that takes two integers and subtracts them together:

subTwo:: Int -> Int -> Int

The parameters are separated with the -> where the return type is the last item.

The parameters are separated with the -> where the return type is the last item.

Some of the common types are as follows:

- **Int**: Int stands for integer. Integers are the basic numeric built-in data types. Plain integer numeric values can be positive or negative whole numbers. On 32-bit machines, the maximum possible Int is 2147483647 and the minimum is -2147483648.
- **Integer**: Integer is a signed integer of unbounded size. Integers are not used for Ints because they are more expensive, both in performance and space consumption.
- **Float**: Float numbers are the real numbers, which contain a fractional part along with a whole number. These numbers include a decimal point (.) to divide the whole and fractional part.
- **Double**: Double is a real floating point that has double the precision.
- **Bool**: Bool is a Boolean type. It can have only two values: True and False.
- **Char**: Char stands for character. It is denoted by single quotes.

6.3.7 Composite Data Type

There are some types which are constructed from other types. They are known as composite data type. The most common composite data types in Haskell are lists and tuples.

Let us discuss two functions head and tail. The head function returns the first element of a list as shown in Figure 24

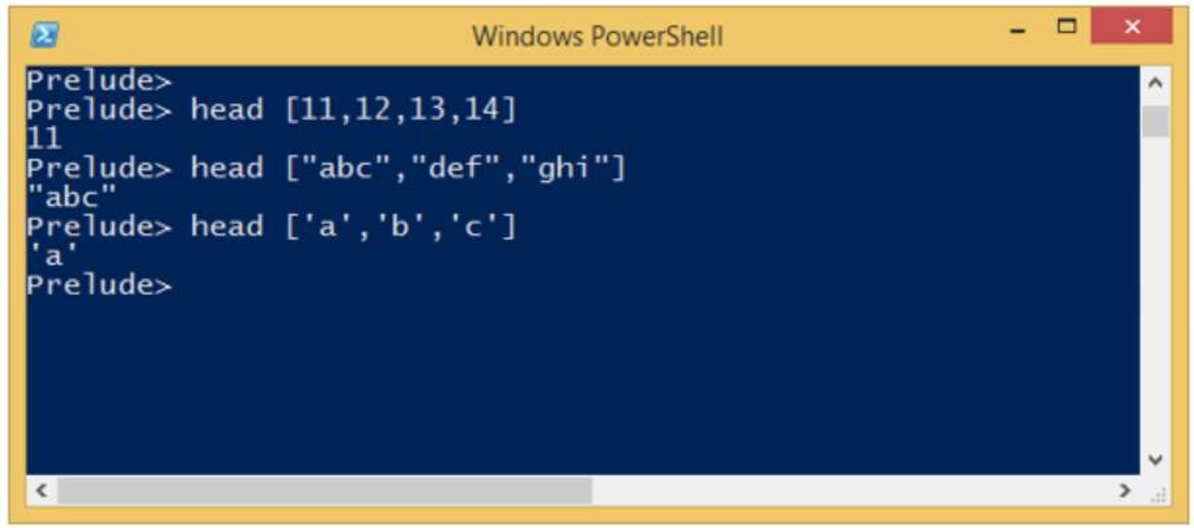


Figure 24: The Head Function

The `tail` function returns everything, but the head of a list as shown in Figure 25:

```
Prelude>
Prelude> tail [11,12,13,14]
[12,13,14]
Prelude> tail ["abc","def","ghi"]
["def","ghi"]
Prelude> tail ['a','b','c']
"bc"
Prelude>
```

Figure 25: The Tail Function

As shown in Figure 24 and 25, the head and tail function can be applied to list of various types. If the head function is applied to a [Char] value, it will return a Char value. The type of the list does not matter.

Since the value of the list can be of any type, the list is called as polymorphic. A polymorphic type is written using a type variable that begins with a lowercase letter. It is a placeholder which will eventually be substituted as a real type.

6.3.8 Typeclasses

A typeclass is a generic interface that is used to define some behavior. It defines a set of functions that has different implementations depending on the type of data. If a type is a part of a typeclass, then it means that it supports and implements the behavior of the typeclass.

Figure 26 shows the type of (==):

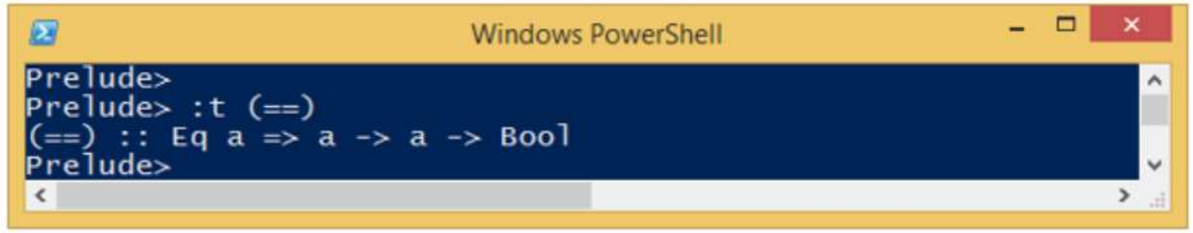
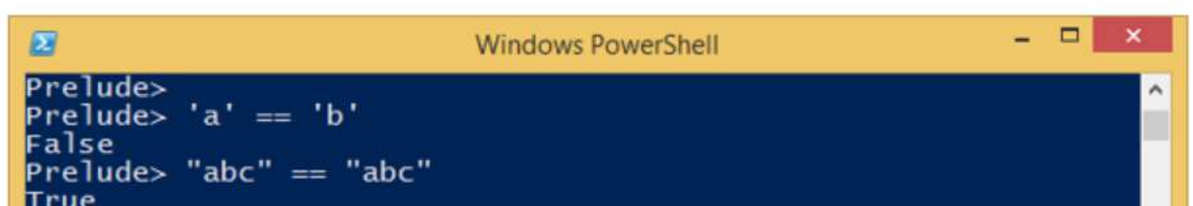


Figure 26: The type of (==)

As shown in Figure 26, the (==) is a function and a new symbol => is displayed. This symbol is known as a class constraint. The output can be read as "For all types a, so long as a is an instance of Eq, (==) takes two parameters of type a and returns a Bool." The Eq typeclass is used to provide an interface that tests for equality. All the standard Haskell types except for IO and functions are a part of the Eq typeclass.

Some basic typeclasses are as follows:

Eq: Eq typeclass is used for the types that can test for equality. Its members implement == and /= functions. Whenever you check any quality between the standard types, then you make the call to the Eq class. Figure 27 shows the examples of Eq typeclass



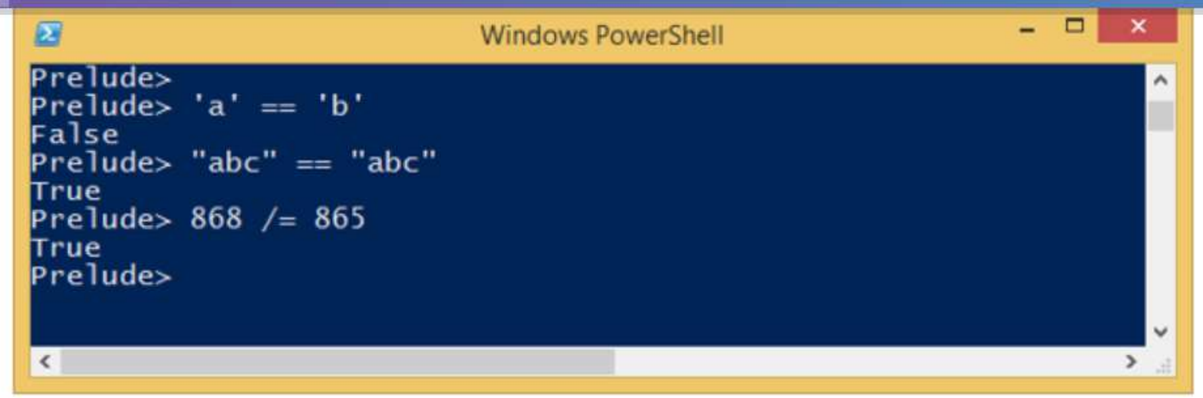
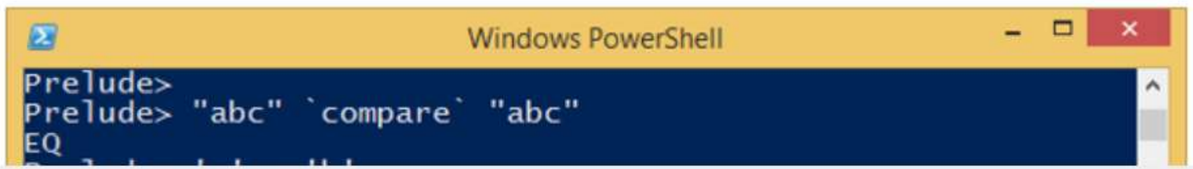


Figure 27: Examples of Eq Typeclass

- **Ord**: Ord typeclass is used for the types that provide the functionality of ordering. Its members implement >, <, >=, <= and compare functions. The compare function is used to compare two members of the same type and returns an ordering. To be a member of Ord typeclass, it should be a member of Eq typeclass. Figure 28 shows the examples of Ord typeclass:



members implement >, <, >=, <= and compare functions. The compare function is used to compare two members of the same type and returns an ordering. To be a member of Ord typeclass, it should be a member of Eq typeclass. Figure 28 shows the examples of Ord typeclass:

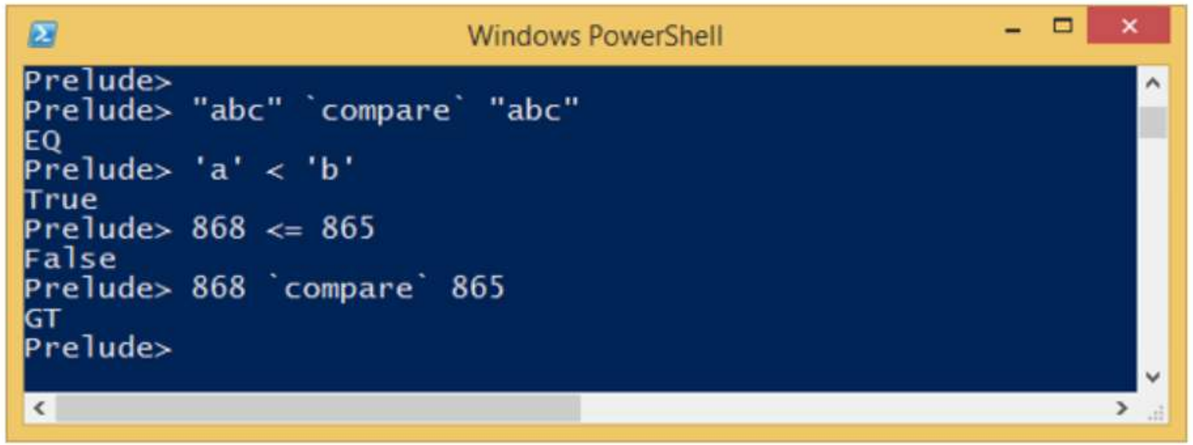


Figure 28: Examples of Ord Typeclass

- **Show**: Show typeclass is used for the types that provide the functionality of print their arguments as string. The most commonly used function which deals with the Show typeclass is show. It takes a value whose type is a member of Show and print it as a string. Figure 29 shows the examples of show function:

Type here to search
ENG
22:31
22-11-2020

SYBGEN ebooks Portal › Course
ebooks.sybgenlearning.com/appLms/index.php?modname=organization&op=custom_enditem&id_item=82

Figure 28: Examples of Ord Typeclass

- **Show**: Show typeclass is used for the types that provide the functionality of print their arguments as string. The most commonly used function which deals with the Show typeclass is show. It takes a value whose type is a member of Show and print it as a string. Figure 29 shows the examples of show function:

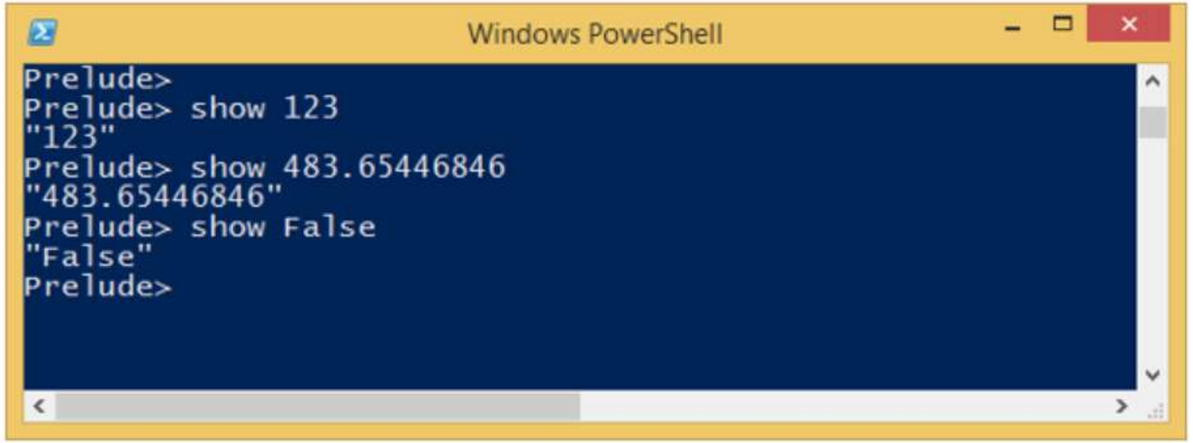


Figure 29: Examples of show Function

Type here to search
ENG
22:32
22-11-2020

Figure 29: Examples of show Function

- **Read**: Read is kind of opposite of Show. The read function takes a string and returns a type which is a member of Read. Figure 30 shows the examples of read function:

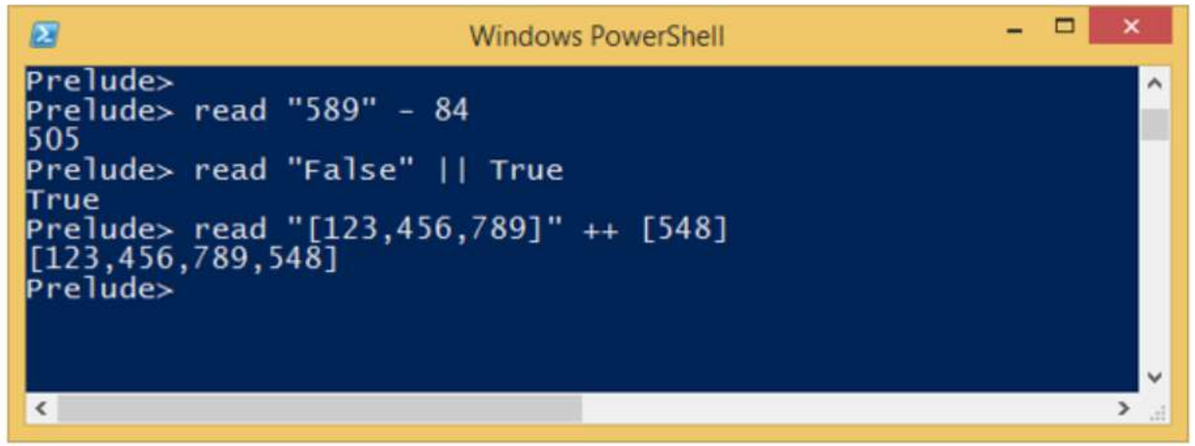


Figure 30: Examples of read Function

- **Enum**: Enum typeclass provides the sequential or ordered functionality. Its advantage is that its types can be used in list ranges. The succ and pred functions are used to get the successors and predecessors. The types that belong in this class are (), Bool, Char, Ordering, Int, Integer, Float and Double. Figure 31 shows the examples of succ and pred functions:

- **Enum**: Enum typeclass provides the sequential or ordered functionality. Its advantage is that its types can be used in list ranges. The succ and pred functions are used to get the successors and predecessors. The types that belong in this class are (), Bool, Char, Ordering, Int, Integer, Float and Double. Figure 31 shows the examples of succ and pred functions:

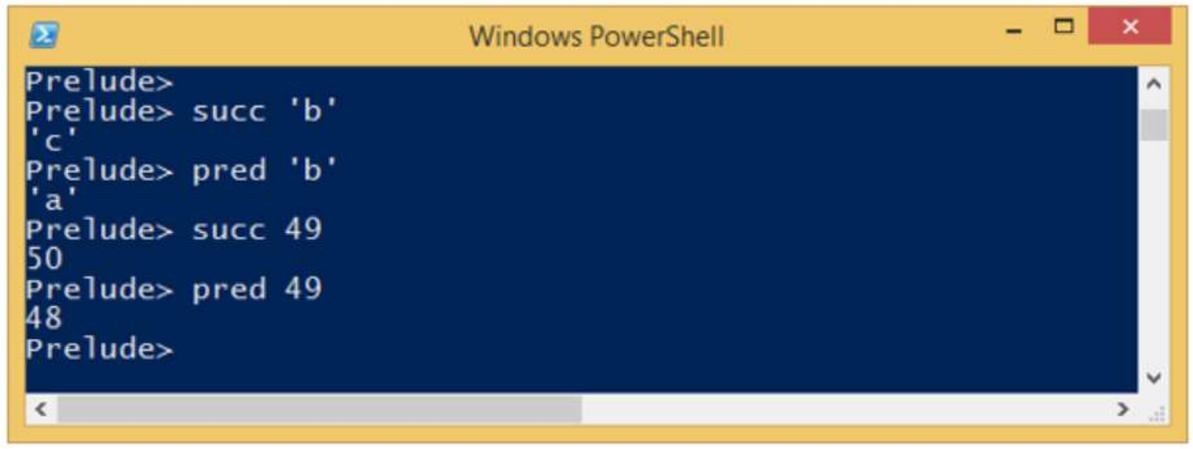


Figure 31: Examples of succ and pred Functions

- **Bounded**: The members of the Bounded typeclass have an upper and a lower bound. The upper and lower bound can be found out using the maxBound and minBound functions. A tuple is also considered as the part of the Bounded typeclass if its components are a part of it. Figure 32 shows the examples of minBound and maxBound functions:

minBound functions. A tuple is also considered as the part of the Bounded typeclass if its components are a part of it. Figure 32 shows the examples of minBound and maxBound functions:

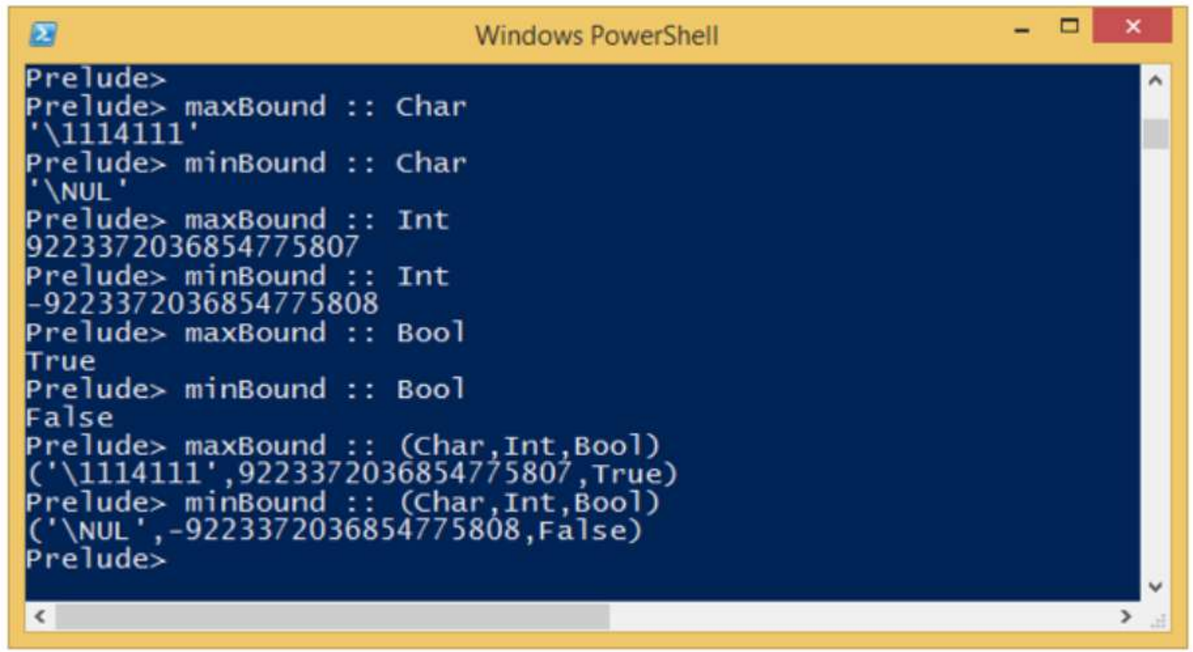


Figure 32: Examples of minBound and maxBound Functions

```
Prelude> minBound :: (Char,Int,Bool)
('\NUL',-9223372036854775808,False)
Prelude>
```

Figure 32: Examples of minBound and maxBound Functions

- **Num**: Num typeclass is a numeric typeclass whose members can act like numbers. The whole numbers can act like any member of the Num typeclass. To be in Num typeclass, they should be a part of the Show and Eq typeclasses. Figure 33 shows the examples of Num typeclass:

Windows PowerShell

```
Prelude>
Prelude> 56 :: Int
56
Prelude> 56 :: Integer
56
Prelude> 56 :: Float
56.0
Prelude> 56 :: Double
56.0
Prelude>
```

Figure 33: Examples of Num Typeclass

```
Prelude>
```

Figure 32: Examples of minBound and maxBound Functions

- **Num**: Num typeclass is a numeric typeclass whose members can act like numbers. The whole numbers can act like any member of the Num typeclass. To be in Num typeclass, they should be a part of the Show and Eq typeclasses. Figure 33 shows the examples of Num typeclass:

Windows PowerShell

```
Prelude>
Prelude> 56 :: Int
56
Prelude> 56 :: Integer
56
Prelude> 56 :: Float
56.0
Prelude> 56 :: Double
56.0
Prelude>
```

Figure 33: Examples of Num Typeclass

6.4 Functions in Haskell

Earlier you have learned about only predefined functions. Now, let's move on to writing your own functions. Let us start with a simple function that adds two numbers that has the following code:

```
add :: Integer -> Integer -> Integer

add a b =  a + b

main = do

  putStrLn "Addition of Two Numbers:"

  print(add 8 3)
```

The first line is the function declaration in which the types of inputs and outputs are described. The second line is the function definition. The third line consists of the main function from which Haskell begins to compile the code. The result of the aforementioned code is as follows:

```
Addition of Two Numbers:

11
```


Establishing secure connection...
Type here to search
ENG
22:32
22-11-2020

In this section, you will learn about pattern matching, guards, clauses, higher-order functions and lambda expressions used in functions.

6.4.1 Haskell Files

Haskell source files are usually identified with a suffix of .hs. In order to create a Haskell file, open a text editor, write the code and save the file with an .hs extension. To load the source file into GHCi, first change the current directory to the directory that contains the .hs file using the :cd command. Suppose you have saved the Haskell source files in the path D:\Haskell, then change the directory.

Suppose there is a source file in the folder named as add.hs. This file contains the code of addition to two number discussed earlier. This file can be loaded into GHCi using the :load or :l command as shown in Figure 34:

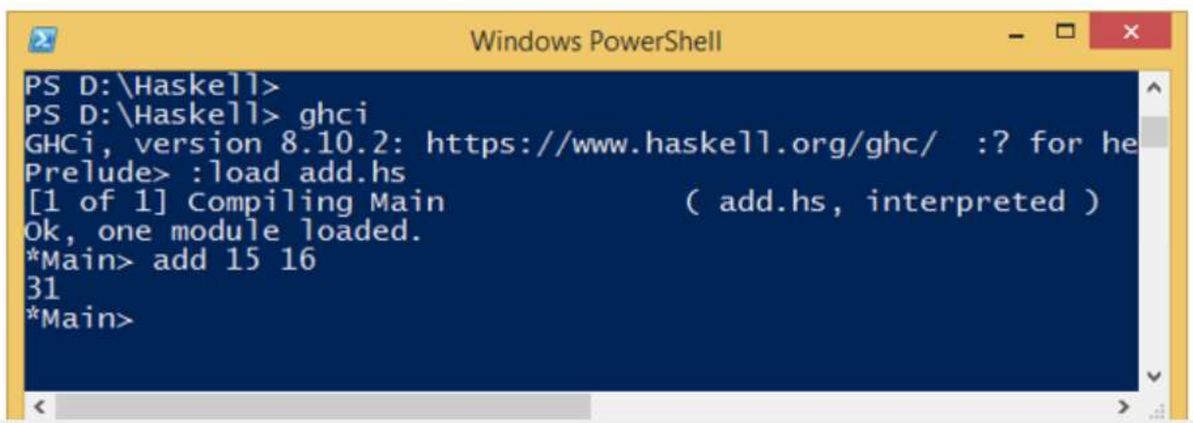


Figure 34: Loading Files

As shown in Figure 34, `Prelude>` changes to `*Main>`. If the source file is modified, it needs to be saved and then reloaded using the `:reload` command, as shown in Figure 35:

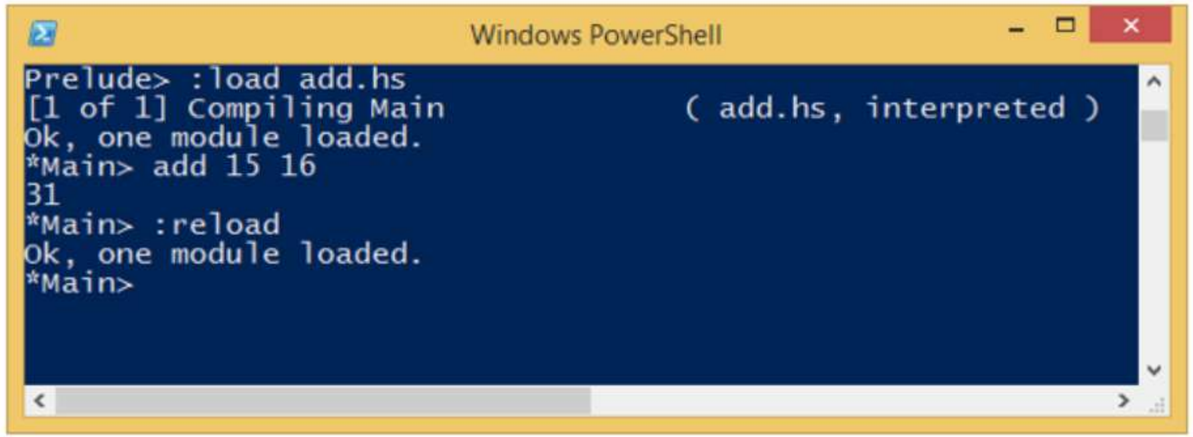


Figure 35: Reloading File

In order to import external modules, such as modules from libraries, the import command is used as shown in Figure 36:

```
PS D:\Haskell>
PS D:\Haskell> ghci
GHCi, version 8.10.2: https://www.haskell.org/ghc/  :? for help
Prelude> import Data.Maybe
Prelude Data.Maybe>
```

Figure 36: Using import Command

Modules can be imported in other modules using the following syntax:

```
module Module_Name where

    import Module1

    import Module2

...
```

Similar to other programming languages, Haskell has a function called main, which specifies that an I/O action will be performed. Create the `main.hs` file and add the following code:

```
main = putStrLn "Programming in Haskell"
```

Now, save the file, load and execute it. Figure 37 shows its output:

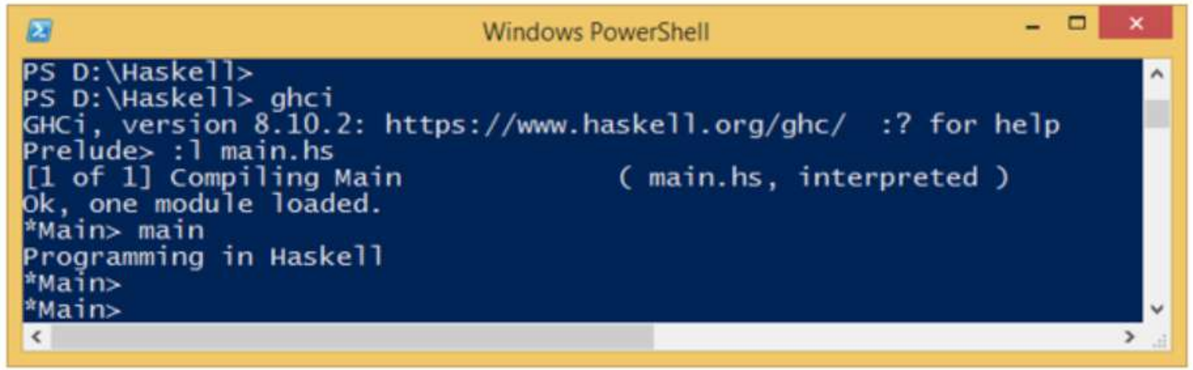


Figure 37: Output of main.hs

In order to write more I/O actions in main, the do block is used. Open the main.hs file and replace the code with the following code:

```
main = do
    putStrLn "Do you like Haskell?"
    answer <- getLine
    putStrLn ("Your answer was: " ++ answer)
```

Now, save the file, load and execute it. Figure 38 shows its output:

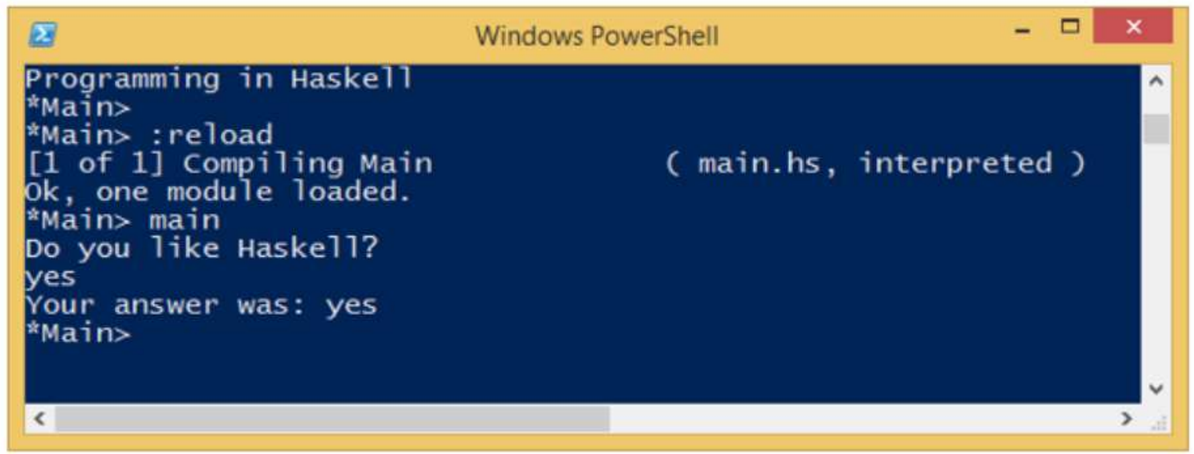


Figure 38: Output of main.hs

6.4.2 Pattern Matching

SYBGEN ebooks Portal › Course
ebooks.sybgenlearning.com/appLms/index.php?modname=organization&op=custom_enditem&id_item=82

6.4.2 Pattern Matching

Pattern matching means to check if some data matches a certain pattern or a type of expression and then acts accordingly. It can be applied to any data type. A function can have different bodies for different patterns. Let's check the following code:

```
month :: (Integral a) => a -> String

month 1 = "January"

month 2 = "February"

month 3 = "March"

month 4 = "April"

month 5 = "May"

month 6 = "June"

month 7 = "July"

month 8 = "August"

month 9 = "September"
```


Type here to search ENG 22:32 22-11-2020

month 12 = "December"

month x = "A year has only 12 months!"

Now, save the file, load and execute it. Figure 39 shows its output:

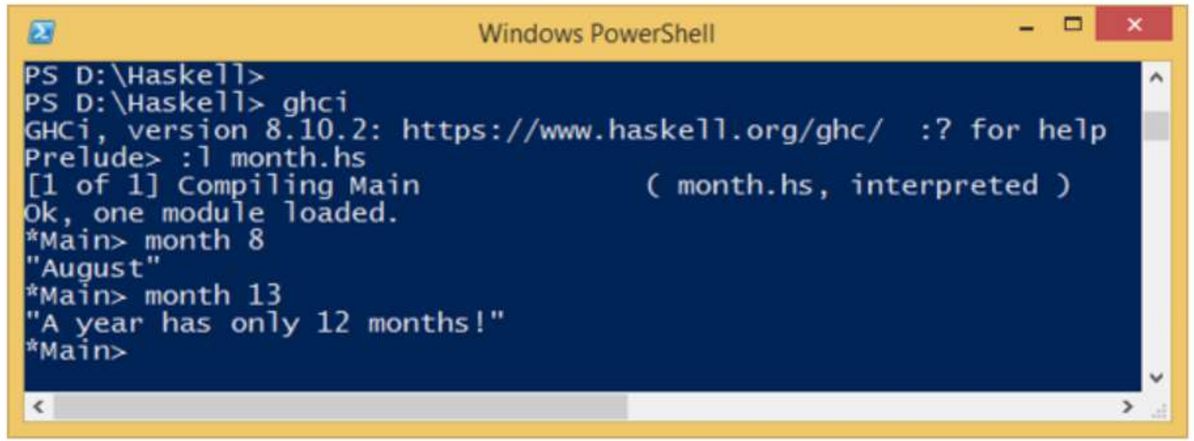


Figure 39: Output of month.hs

When month is called, the matching is started. When there is a match, the corresponding body is chosen. The function also has a default pattern. If there is no default pattern, then an exception will be displayed. This function can also be written using an if..then..else statement.

Consider a function safetail that behaves in the same way as tail, except that safetail maps the empty list to the empty list, whereas tail gives an error in this case.

Consider a function safetail that behaves in the same way as tail, except that safetail maps the empty list to the empty list, whereas tail gives an error in this case. Define safetail using:

- Conditional expression

safetail xs = if xs == [] then [] else tail xs

- Guarded equations:
- safetail' xs | xs == [] = []

  | otherwise = tail xs

- Pattern matching:

safetail'' [] = []

safetail'' xs = tail xs

6.4.3 Case Expressions

The case construct is also used to match patterns within an expression, which is followed by an arbitrary expression. The pattern match is performed against the result of this expression.

6.4.3 Case Expressions

The case construct is also used to match patterns within an expression, which is followed by an arbitrary expression. The pattern match is performed against the result of this expression.

The “of” keyword signifies the end of the expression and the beginning of the block of patterns and expressions.

Every item in the block consists of a pattern followed by an arrow (->), then followed by an expression to evaluate whether that pattern matches. It is necessary that these expressions are of same type. The result is the result of the expression associated with the first pattern to match. Matches are performed from top to bottom. In case none of the other patterns match, then the wild card pattern _ is used as the last in the list of patterns.

The general syntax of a case expression is:

```
case expression of pattern -> result
                   pattern -> result
                   ...
```

The following code shows the month function using case:

```
month :: (Integral a) => a -> String
```

```
1 -> "January"
2 -> "February"
3 -> "March"
4 -> "April"
5 -> "May"
6 -> "June"
7 -> "July"
8 -> "August"
9 -> "September"
10 -> "October"
11 -> "November"
12 -> "December"
_ -> "A year has only 12 months!"
```

12 -> "December"

_ -> "A year has only 12 months!"

Now, save the file, load and execute it. Figure 40 shows its output:

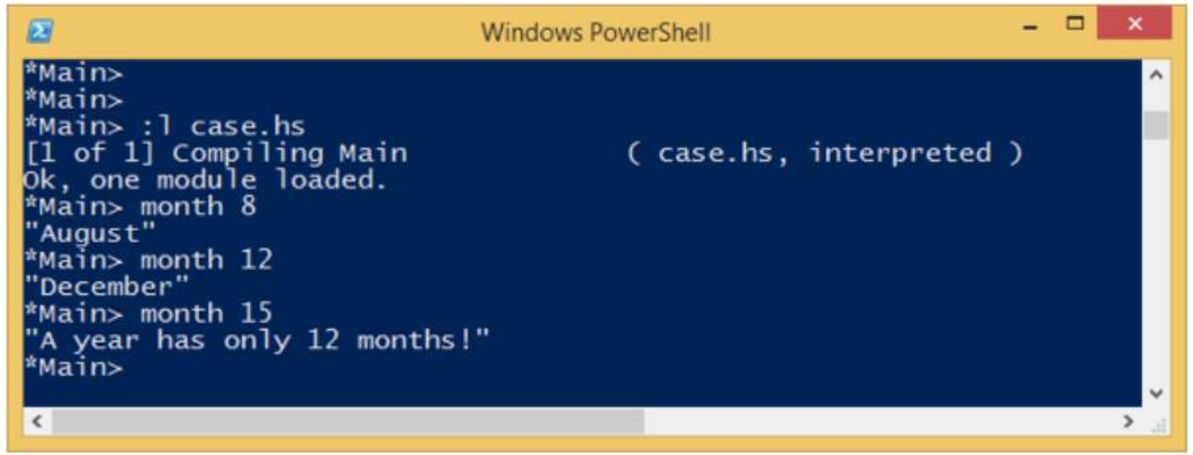


Figure 40: Output of case.hs

6.4.4 Guards

Patterns are used to ensure that a value matches to some form and guards are a way to test if the value has some property or not. Guards help in performing fixed tests

6.4.4 Guards

Patterns are used to ensure that a value matches to some form and guards are a way to test if the value has some property or not. Guards help in performing fixed tests of a value's shape. They help in testing whether a value has a certain property.

A pattern can be followed by zero or more guards where each is an expression of type Bool. A guard is introduced by a | symbol. This is followed by the guard expression and then an = symbol (or -> in case of a case expression). Then body is specified to use if the guard expression evaluates to True. If the pattern is matched, each guard associated with that pattern is evaluated in the order in which they are written. If a guard is succeeded, the body affiliated with it is used as the result of the function. If no guard succeeds, pattern matching moves on to the next pattern.

The following code is used to check if a number is positive or negative:

```
sign :: (RealFloat a) => a -> String
sign x
 | x < 0     = "negative"
 | x == 0    = "zero"
 | otherwise = "positive"
```

Now, save the file, load and execute it. Figure 41 shows its output:

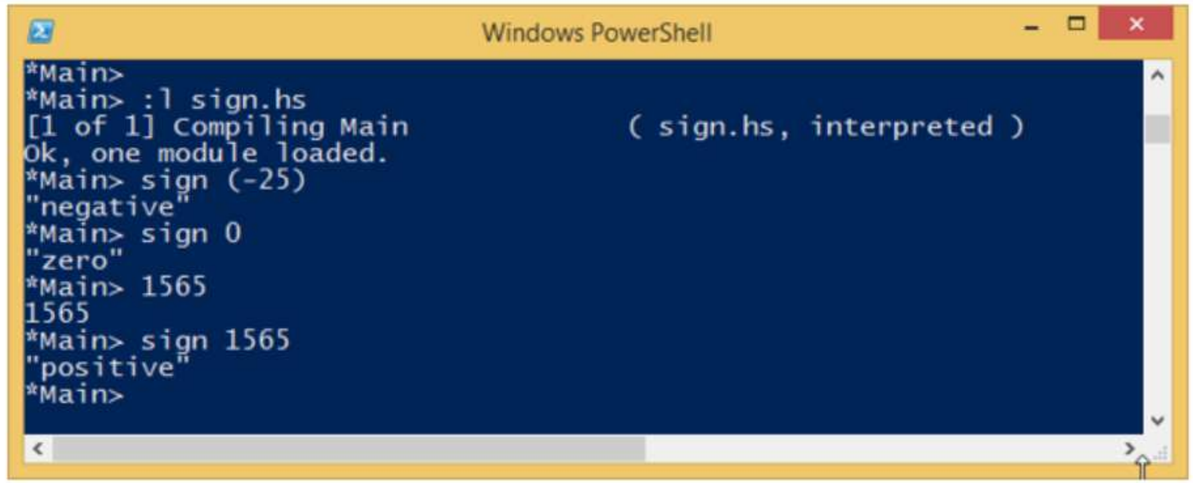


Figure 41: Output of sign.hs

6.4.5 The where and let Clauses

In Haskell, there is no way to define immediate variables. The expressions that are used come from the patterns of the arguments. Haskell does not have variables because the value of the expressions is fixed. Let us consider the following code where various math operations are performed on inputs:

mathCalc :: Int -> Int -> Int -> Int

```
mathCalc :: Int -> Int -> Int -> Int

mathCalc x y z = (z - x) + (y - x) + (x * y * z) + x
```

The mathCalc function defined in the aforementioned code is one line but not very readable. It can be made more readable using the intermediate expressions. Let's first perform this using a where clause:

```
mathCalc:: Int -> Int -> Int -> Int

mathCalc x y z = diff1 + diff2 + prod + x

  where

    diff1 = z - x

    diff2 = y - x

    prod = x * y * z
```

In the aforementioned code, the where section declares diff1, diff2 and prod as intermediate values. It is possible to use them in the base of the function and use their results within each other, as shown in the following code:

```
mathCalc:: Int -> Int -> Int -> Int

mathCalc x y z = diff1 + diff2 + prod + x
```

```
prod = diff2 * x * y

diff1 =  y - z

diff2 = z - diff1
```

However, it is necessary to ensure that there are no loops in the subexpressions.

The aforementioned function can also be implemented using a let statement. It has similar syntactic construct, except the new expressions are declared before. Then the in keyword is used to signal the expression that will use the values, as shown in the following code:

```
mathCalc:: Int -> Int -> Int -> Int

mathCalc x y z =

  let diff1 = z - x

    diff2 = y - x

    prod = x * y * z

  in diff1 + diff2 + prod + x
```

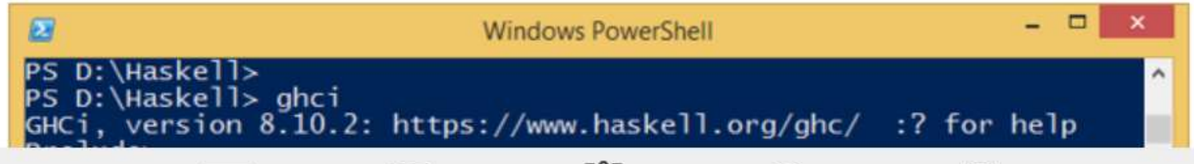
In I/O cases, the let statement can be used without the in keyword. The let statement is used instead of where when the expression depends on the user's input, as

```
main :: IO ()
main = do
  input <- getLine
  let repeated = replicate 3 input
  print repeated
```

6.4.6 Infix Functions

When a function is defined or applied in Haskell, the name of the function is written function followed by its arguments. This notion is referred to as prefix as the name of the function comes before its arguments.

If a function takes two or more arguments then it can be used in it in infix form where the function is placed between its first and second arguments. In order to define a function using infix notation, the function's name is enclosed in backtick (`) characters. Figure 42 shows the definition and implementation of an infix function:



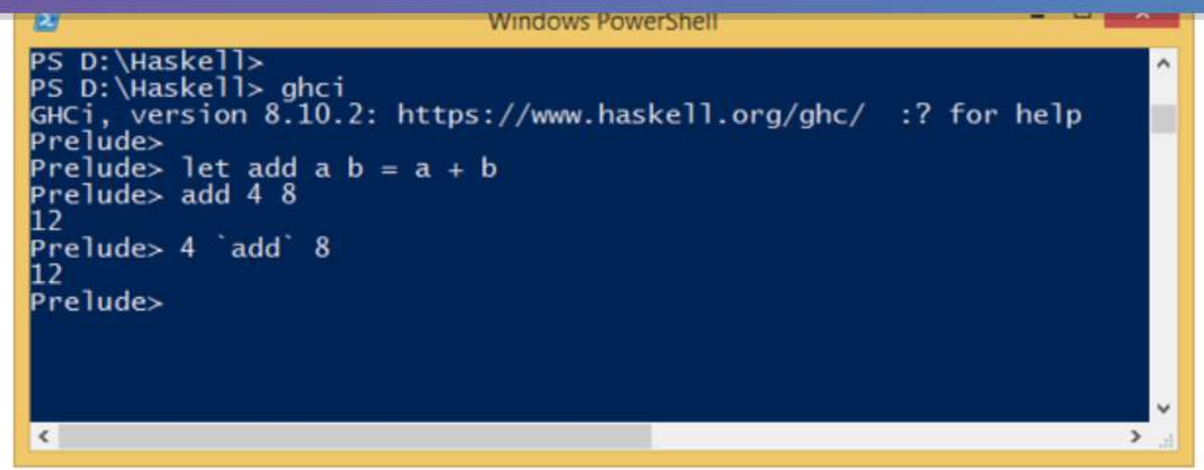


Figure 42: Implementation of Infix Function

6.4.7 Introduction to Lambda Calculus

Haskell also allows you to completely write anonymous functions that can be used to avoid giving names to the helper functions. Anonymous functions are known as "lambda" functions. They are introduced with a backslash character (\) pronounced lambda followed by the function's arguments and then an arrow (->) to introduce the function's body.

Anonymous functions behave in a similar manner to the named functions. However, there are some restrictions on the way of defining them. A normal function can be

6.4.7 Introduction to Lambda Calculus

Haskell also allows you to completely write anonymous functions that can be used to avoid giving names to the helper functions. Anonymous functions are known as “lambda” functions. They are introduced with a backslash character (\) pronounced lambda followed by the function’s arguments and then an arrow (->) to introduce the function’s body.

Anonymous functions behave in a similar manner to the named functions. However, there are some restrictions on the way of defining them. A normal function can be written using multiple clauses containing different patterns and guards. However, a lambda can only have a single clause in its definition. This restricts the usage of patterns in the definition of a lambda. The following code shows the usage of lambda:

```
main = do
  putStrLn "The cube of 27 is:"
  print ((\x -> x^3) 27)
```

In the aforementioned code, the expression x -> x^3 is defined and called for the value of 27. Now, save the file, load and execute it. Figure 21 shows its output:

patterns in the definition of a lambda. The following code shows the usage of lambda:

```
main = do
  putStrLn "The cube of 27 is:"
  print ((\x -> x^3) 27)
```

In the aforementioned code, the expression x -> x^3 is defined and called for the value of 27. Now, save the file, load and execute it. Figure 21 shows its output:

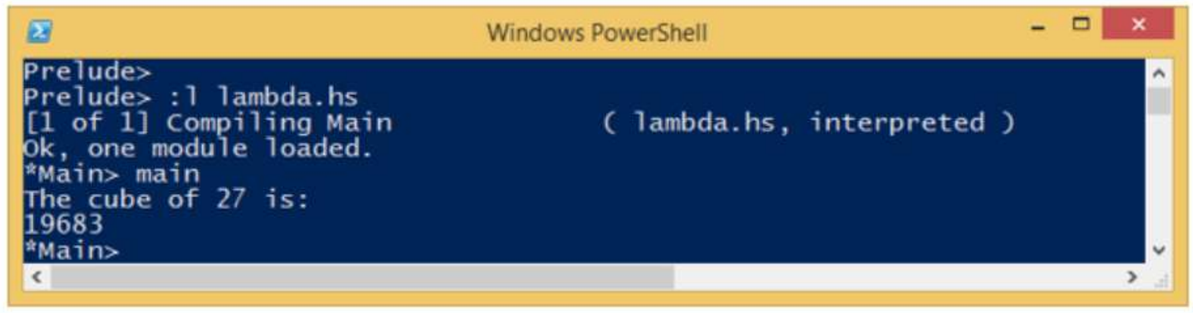


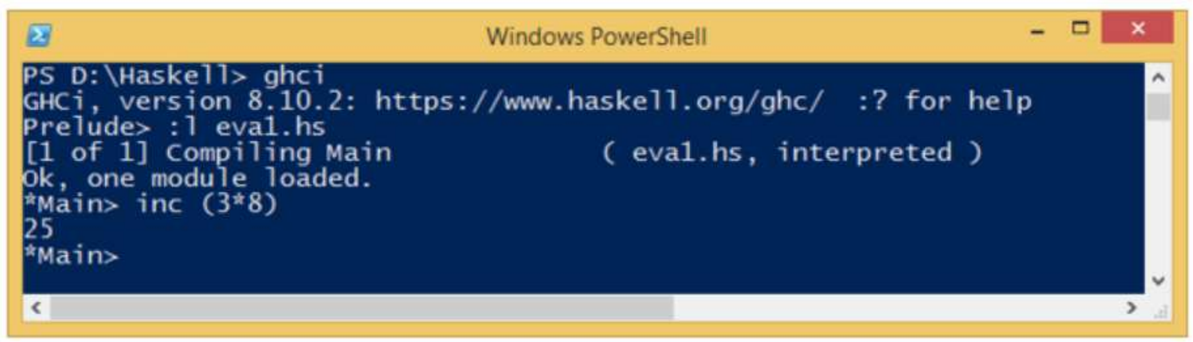
Figure 43: Output of lambda.hs

6.5 Evaluation order

In this section, you will learn about the lazy evaluation. It is the mechanism that is used to evaluate the expressions in Haskell. The common methods of computation in Haskell is the application of functions to the arguments. For example, if a function is defined that increments an integer:

```
inc :: Int -> Int

inc n = n + 1
```

Figure 44 shows the output of this function:



SYBGEN ebooks Portal › Course
ebooks.sybgenlearning.com/appLms/index.php?modname=organization&op=custom_enditem&id_item=82

6.5 Evaluation order

It is evaluated in the following manner:

```
inc (3*8)

//{applying *}

= inc 24

//{applying inc}

= 24+1

//{applying +}

=25
```

It can still be obtained by performing the first two function applications in the opposite order:

inc (3*8)

{applying inc}

= (3*8) + 1

{applying *}

Waiting for admin.kotobee.com...
Type here to search
ENG
22:33
22-11-2020

= 24 + 1

{applying +}

= 25

When the order of the application is changed, the final result does not affect to only those examples. It is a general property of the function applications in Haskell. In Haskell, two expressions will evaluate and provide the same value. It is not possible for the imperative programming languages.

6.5.1 Evaluation Strategies

An expression that is used in a form of a function and applied to one or more arguments that can be “reduced” by performing the application is called a reducible expression. It is also known as redex for short. These reductions do not decrease the size of an expression. Suppose we have defined a function div that takes a pair of integers and returns their product:

div :: (Int, Int) -> Int

div (x, y)=x / y

Now, let us consider an expression div (12 + 12, 4 + 4). This expression contains three redexes that are 12 + 12 and 4 + 4, which have the form of the function + applied to two arguments, and the entire expression div (12 + 12, 4 + 4) itself. If you perform the reductions, the expressions will be div (24, 4 + 4), div (12 + 12, 8) and (12 +

SYBGEN ebooks Portal › Course
ebooks.sybgenlearning.com/appLms/index.php?modname=organization&op=custom_enditem&id_item=82

6.5 Evaluation order

Now, let us find out the order in which the reductions should be performed. A common strategy is the innermost evaluation that selects the redex that is innermost. If there are more than one innermost redex, then you select the one that begins at the leftmost position in the expression. For example, both (12 + 12) and (4 + 4) contain no other redexes. So, they are the innermost. Therefore, you will start with the redex (12 + 12) beginning at the leftmost position.

Innermost evaluation can also be characterised in terms of how arguments are passed to functions. This strategy ensures that the function is always evaluated before it is applied. The arguments are passed by value.

Another common strategy is dual to innermost evaluation that always selects the redex that is outermost. The evaluation procedure for this expression will be:

div (12 + 12, 4 + 4)

//{applying div}

=(12 + 12)/(4 + 4)

//{applying the first +}

= 24 / (4 + 4)

{applying +}

= 24 / 8

{applying /}

Type here to search ENG 22:34 22-11-2020

= 3

6.5.2 I/O in Haskell

In this section, you will learn how to write a basic Haskell program. Open a text editor and write the following code:

main = putStrLn “Hello, it is fun learning Haskell!”

In this program, a name called main is defined and a putStrLn function is defined with some parameter. Save the file as hello.hs.

Open the Windows PowerShell and navigate to the directory where hello.hs is stored and perform the following command:

ghc –make hello

Now, run the program by typing the command ./hello. Figure 45 shows the output:

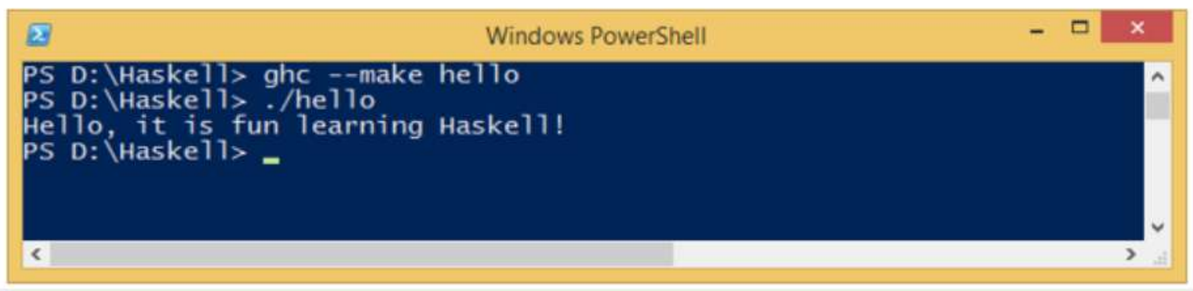


Figure 45: Output of hello.hs

Now, if you will check the type of the function putStrLn, you will find the output similar to the one shown in Figure 46:

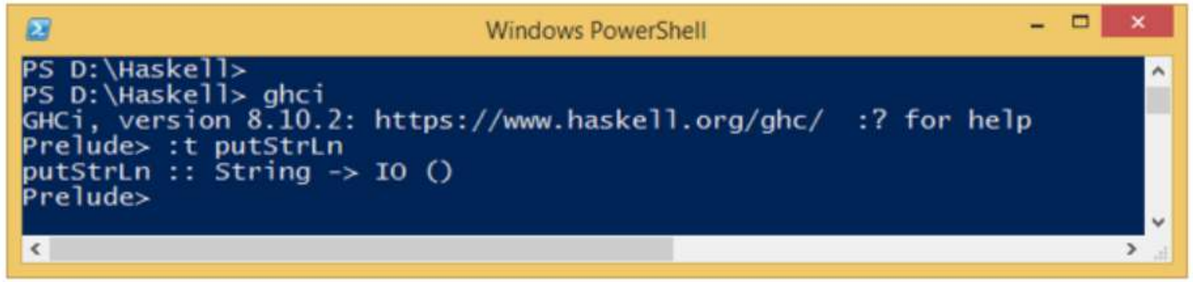


Figure 46: Type of putStrLn Function

As you can see in Figure 46, the putStrLn function takes a string and returns an I/O action that has a result type of (). An I/O action is something which is when performed performs an action with an effect and also contains some return value. An I/O action is performed only when a name of main is given to the actions. The <- operator pulls out the result from performing an I/O action and store it in a variable.

Let's create a program using do syntax that combines several I/O actions into one. The code of the program is as follows:

```
main = do
  putStrLn "Hello, which programming language are you studying"
```

```
language <- getLine

putStrLn ("Yes, you are studying " ++ language ++ " language.")
```

In the aforementioned code, each step is an action that are connected using the do syntax. It uses the getLine action that contains a result of type String. It waits for the user's input and then binds the input to language.

Some of the common actions that are useful with dealing I/O are listed in Table 3:

Table 3: Functions with Definitions

Functions	Definitions
`putStr`	Takes a string as a parameter and returns an I/O action that will print that string to the terminal.
`putChar`	Takes a character and returns an I/O action that will print it out to the terminal.
`print`	Takes a value of any type that's an instance of show, calls show with that value to stringify it and then outputs that string to the terminal.
`getChar`	Reads a character from the input.

	value to stringify it and then outputs that string to the terminal.
`getChar`	Reads a character from the input.
`sequence`	Takes a list of I/O actions and returns an I/O actions that will perform those actions one after the other.
`forever`	Takes an I/O action and returns an I/O action that just repeats the I/O action it got forever.

6.5.3 Files and Handles

Haskell defines some basic functions for I/O that are listed in the System.IO library. One common function is openFile that provides a file Handle. This Handle is then used to perform some operations on the file. Haskell also provides functions, such as hPutStrLn, which is similar to putStrLn, but takes an additional argument a Handle that specifies the files. If you want to close the Handle, use hClose.

The following code is an example that displays the contents of this file on the console:

```
main = do
```

SYBGEN ebooks Portal › Course
ebooks.sybgenlearning.com/appLms/index.php?modname=organization&op=custom_enditem&id_item=82

The following code is an example that displays the contents of this file on the console:

```
main = do
  let file = "read.txt"
  data <- readFile file
  putStrLn data
```

Some of the common functions that are useful with dealing files are listed in Table 4:

Table 4: Functions with Definitions

Functions	Definitions
`hGetContents`	Takes a Handle, so it knows which file to get the contents from and returns an IO String.
`hClose`	Takes a handle and returns an I/O action that closes the file.
`hPutStrLn`	Takes a handle and a string and returns an I/O action that will write that string to the file associated with the handle and then put a newline after it.

Type here to search
ENG
22:34
22-11-2020

SYBGEN ebooks Portal › Course
ebooks.sybgenlearning.com/appLms/index.php?modname=organization&op=custom_enditem&id_item=82

`hPutStrLn`	Takes a handle and a string and returns an I/O action that will write that string to the file associated with the handle and then put a newline after it.
`readFile`	Takes a path to a file and returns an I/O action that will read that file and bind its contents.
`writeFile`	Takes a path to a file and a string to write to that file and returns an I/O action that will do the writing.
`hFlush`	Takes a handle and returns an I/O action that will flush the buffer of the file associated with the handle.
`openTempFile`	Takes a path to a temporary directory and a template name for a file and opens a temporary file.

6.5.4 Monads and Monoids in Haskell

A monad is defined as a way for values that can be used in sequences of computations, resulting in a structure of computations. The sequential building blocks can be

Type here to search ENG 22:34 22-11-2020

6.5.4 Monads and Monoids in Haskell

A monad is defined as a way for values that can be used in sequences of computations, resulting in a structure of computations. The sequential building blocks can be used to create computations and the building blocks can themselves be structured as computations. Monad is considered as a strategy to combine computations into complex computations.

The following three elements are used to characterise a monad:

- A type constructor m
- A return function, which returns values of the type m
- A binding operation (>>=), which combines the values of type m with computations that output values of type m

A general representation of a monad is as follows:

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> a -> m b -> m b
```

In the first line, the type of the monad is m. In the second line, the value a is taken by the return function and is embedded into the monad m. In the third line, the binding function takes the monad instance m a and a computation that creates monad instance m b from a's produces the new monad instance m b.

6.5 Evaluation order

- **Right unit**: This rule states the following statement:

f >>= return ≡ f

In this statement, f stands for computation. According to this rule, the return function does not change the value and it should also not change anything in the monad.

- **Left unit**: This rule states the following statement:

return a >>= f ≡ f a

In this statement, f stands for computation and a is a value. According to this rule, the return function does not change the value and it should also not change anything in the monad.

- **Associativity**: This rule states the following statement:

f >>= (\x -> g x >>= h) ≡ (f >>= g) >>= h

According to this rule, when you make computations it does not matter how they are grouped, the result will be same in all scenarios.

Monoids

In Haskell, Monoid typeclass is a class that have an operation for combining values with a value that does not do anything when combined with others. This value is

Monoids

In Haskell, Monoid typeclass is a class that have an operation for combining values with a value that does not do anything when combined with others. This value is known as the identity element.

A monoid consists of two things – a 'unit' value known as mempty and an append operation that combines values known as mappend. A general representation of Monoid typeclass is as follows:

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
```

The mappend is usually referred using the (<>) operator. It is used to merge together a sequence of items and results in the same value regardless of the evaluation order. It should follow the following laws:

a <> (b <> c) = (a <> b) <> c

mempty <> a = a

a <> mempty = a

6.5.5 Recursion in Haskell

a <> mempty = a

6.5.5 Recursion in Haskell

In the previous section, you learned about functions. In real-world applications, most of the time, the functions recall themselves which is known as recursion. Recursion is a way of programming a problem in which a function calls itself one or more times, breaking down a problem into smaller subproblems. The breakdown continues until a problem, which is small enough and can be solved trivially, is not obtained. In simpler words, through recursion, one can write swift and elegant solutions to problems that may otherwise be very complex and complicated to code.

Let us discuss a simple example, the factorial function is defined as follows:

This is also known as n!. The first condition is the edge condition which defines the condition for function termination. This helps in finishing the recursive call, otherwise it will enter into an infinite loop.

This factorial function in Haskell can be written in the following manner:

```
fact 0 = 1

fact n = n * fact (n - 1)
```

This is also known as n!. The first condition is the edge condition which defines the condition for function termination. This helps in finishing the recursive call, otherwise it will enter into an infinite loop.

This factorial function in Haskell can be written in the following manner:

fact 0 = 1

fact n = n * fact (n - 1)

Now, save the file as fact.hs, load and execute it. Figure 47 shows its output:

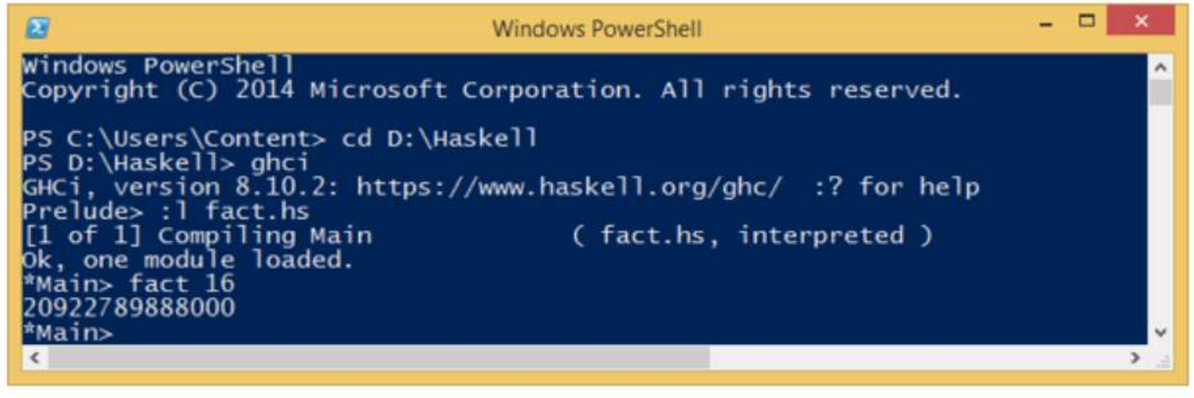


Figure 47: Output of fact.hs

While selecting the function definition, Haskell starts with the first definition and selects the

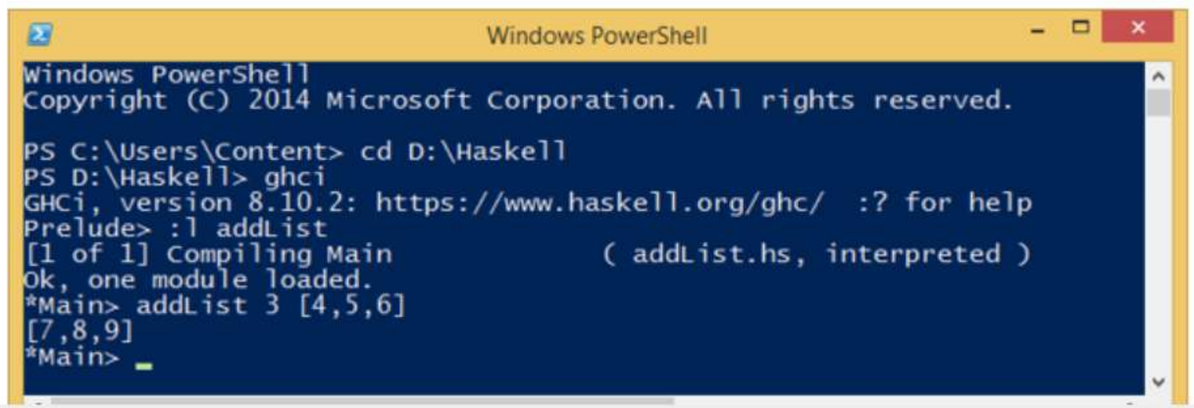
6.6 Higher-order Functions

A higher-order function is defined as a function that takes other functions as arguments or returns a function as result. Let's define the following function:

addList a [] = []

addList a (y:ys) = a+y : addList a ys

Now, save the file, load and execute it. Figure 47 shows its output:



```
*Main>
```

Figure 48: Output of addList.hs

In the aforementioned code, the addList takes two inputs which is a number and list, and the output is a list resulting from adding the number to the elements. A higher-order function is addListby8. Figure 48 shows its implementation:

```
Windows PowerShell
Ok, one module loaded.
*Main> addList 3 [4,5,6]
[7,8,9]
*Main> addListby8 = addList 8
*Main> addListby8 [20,40,60]
[28,48,68]
*Main>
```

Figure 49: A Higher-Order Function

The addListby8 takes one input which is a list as a=8 when addList is called inside addListby8.

6.7 Programming Examples

Example: Write a Haskell snippet to create a list of first 10 numbers

li = [1..10]

OR

li = take 10 [1..]

```
Prelude> li = [1..10]
Prelude> li
[1,2,3,4,5,6,7,8,9,10]
Prelude> li = take 10 [1..]
Prelude> li
[1,2,3,4,5,6,7,8,9,10]
```

Example: Write a Haskell snippet to create a list of first 10 numbers in reverse

li = [10,9..1]

OR

SYBGEN ebooks Portal › Course
ebooks.sybgenlearning.com/appLms/index.php?modname=organization&op=custom_enditem&id_item=82

6.7 Programming Examples

Example: Write a Haskell snippet to create a list of first 10 numbers in reverse

li = [10,9..1]

OR

li = reverse [1..10]

```
Prelude> li = [10,9..1]
Prelude> li
[10,9,8,7,6,5,4,3,2,1]
Prelude> li = reverse [1..10]
Prelude> li
[10,9,8,7,6,5,4,3,2,1]
```

Example: Write a haskell snippet to filter the elements from list whose square root is greater than 7.

li = [12,43,64,314, 13,34,56] (Assumption)

filter (\li -> sqrt li > 7) li

```
Prelude> li = [12,43,64,314, 13,34,56]
Prelude> filter (\li -> sqrt li > 7) li
[64.0,314.0,56.0]
Prelude>
```


Type here to search ENG 22:35 22-11-2020

filter (\li -> sqrt li > 7) li

```
Prelude> li = [12,43,64,314, 13,34,56]
Prelude> filter (\li -> sqrt li > 7) li
[64.0,314.0,56.0]
Prelude>
```

Example: Create powers of 2 as per written in the given list. Also, create a list of squares of the numbers in the given list.

Powers of two:

If li is the given list then,

map (2^) li

will create powers of two and

map (^2) li

will create squares of the given numbers.

```
Prelude> li = [2,4,5,1,6,7]
Prelude> map (2^) li
[4,16,32,2,64,128]
Prelude> map (^2) li
```

6.7 Programming Examples

Powers of two:

If li is the given list then,

map (2^) li

will create powers of two and

map (^2) li

will create squares of the given numbers.

```
Prelude> li = [2,4,5,1,6,7]
Prelude> map (2^) li
[4,16,32,2,64,128]
Prelude> map (^2) li
[4,16,25,1,36,49]
```

Example: Convert marks of student to grades (>75 A grade else B)

Use if then else in the list comprehension.

```
Prelude> li = [55, 79, 80, 34, 89]
Prelude> l = [if i > 75 then "A" else "B" | i <- li]
Prelude> l
["B","A","A","B","A"]
Prelude>
```


Type here to search
ENG
22:35
22-11-2020

Example: Convert marks of student to grades (>75 A grade else B)

Use if then else in the list comprehension.

```
Prelude> li = [55, 79, 80, 34, 89]
Prelude> l = [if i > 75 then "A" else "B" | i <- li]
Prelude> l
["B","A","A","B","A"]
Prelude>
```

Example: Generate a list of all possible triangles with sides having integer values and under or equal to 10.

For a triangle with sides a, b and c to be valid, it is essential to satisfy the given rule

a + b < c

therefore, we can write it as follows:

```
Prelude>triangle = [ [a, b, c] | a<-[1..10] , b<- [1..10], c <- [1..10], (a + b) < c]]
Prelude>
Prelude> triangle
[[1,1,3],[1,1,4],[1,1,5],[1,1,6],[1,1,7],[1,1,8],[1,1,9],[1,1,10],[1,2,4],[1,2,5],[1,2,
6],[1,2,7],[1,2,8],[1,2,9],[1,2,10],[1,3,5],[1,3,6],[1,3,7],[1,3,8],[1,3,9],[1,3,10],[1
,4,6],[1,4,7],[1,4,8],[1,4,9],[1,4,10],[1,5,7],[1,5,8],[1,5,9],[1,5,10],[1,6,8],[1,6,9]
```

or equal to 10.

For a triangle with sides a, b and c to be valid, it is essential to satisfy the given rule

a + b < c

therefore, we can write it as follows:

```
Prelude>triangle = [ [a, b, c] | a<-[1..10] , b<- [1..10], c <- [1..10], (a + b) < c]]
Prelude>
Prelude> triangle
[[1,1,3],[1,1,4],[1,1,5],[1,1,6],[1,1,7],[1,1,8],[1,1,9],[1,1,10],[1,2,4],[1,2,5],[1,2,
6],[1,2,7],[1,2,8],[1,2,9],[1,2,10],[1,3,5],[1,3,6],[1,3,7],[1,3,8],[1,3,9],[1,3,10],[1
,4,6],[1,4,7],[1,4,8],[1,4,9],[1,4,10],[1,5,7],[1,5,8],[1,5,9],[1,5,10],[1,6,8],[1,6,9]
,[1,6,10],[1,7,9],[1,7,10],[1,8,10],[2,1,4],[2,1,5],[2,1,6],[2,1,7],[2,1,8],[2,1,9],[2,
1,10],[2,2,5],[2,2,6],[2,2,7],[2,2,8],[2,2,9],[2,2,10],[2,3,6],[2,3,7],[2,3,8],[2,3,9],
[2,3,10],[2,4,7],[2,4,8],[2,4,9],[2,4,10],[2,5,8],[2,5,9],[2,5,10],[2,6,9],[2,6,10],[2,
7,10],[3,1,5],[3,1,6],[3,1,7],[3,1,8],[3,1,9],[3,1,10],[3,2,6],[3,2,7],[3,2,8],[3,2,9],
[3,2,10],[3,3,7],[3,3,8],[3,3,9],[3,3,10],[3,4,8],[3,4,9],[3,4,10],[3,5,9],[3,5,10],[3,
6,10],[4,1,6],[4,1,7],[4,1,8],[4,1,9],[4,1,10],[4,2,7],[4,2,8],[4,2,9],[4,2,10],[4,3,8]
,[4,3,9],[4,3,10],[4,4,9],[4,4,10],[4,5,10],[5,1,7],[5,1,8],[5,1,9],[5,1,10],[5,2,8],[5
,2,9],[5,2,10],[5,3,9],[5,3,10],[5,4,10],[6,1,8],[6,1,9],[6,1,10],[6,2,9],[6,2,10],[6,3
,10],[7,1,9],[7,1,10],[7,2,10],[8,1,10]]
Prelude>
```

Example: Generate a list of all possible right-angle triangles with sides having integer values and under or equal to 10.

For a triangle with sides a, b and c to be valid, it is essential to satisfy the given rule

6.7 Programming Examples

Example: Generate a list of all possible right-angle triangles with sides having integer values and under or equal to 10.

For a triangle with sides a, b and c to be valid, it is essential to satisfy the given rule

$a^2 + b^2 = c^2$

therefore, we can write it as follows:

rttriangle = [[a, b, c] | a<-[1..10] , b<- [1..a], c <- [1..b], a^2 == b^2 + c^2]

```
Prelude> rttriangle = [ [a, b, c] | a<-[1..10] , b<- [1..a], c <- [1..b], a^2 == b^2 + c^2]
Prelude>
Prelude> rttriangle
[[5,4,3],[10,8,6]]
Prelude>
```

Example: Using inbuilt prefix functions find factorial of a number.

factorial :: Integer -> Integer // can be written if factorial function is defined in .hs file.

factorial n = product [1..n]

```
Prelude> factorial n = product [1..n]
Prelude> factorial 23
```


Type here to search
ENG
22:35
22-11-2020

```
Prelude>
```

Example: Convert day of the week number to week day using pattern matching.

Save the code mentioned below in a file called as “weekday.hs”

```
dayofweek :: (Integral a) => a -> String
dayofweek 0 = "Sunday"
dayofweek 1 = "Monday"
dayofweek 2 = "Tuesday"
dayofweek 3 = "Wednesday"
dayofweek 4 = "Thursday"
dayofweek 5 = "Friday"
dayofweek 6 = "Saturday"
dayofweek x = "Invalid day of week"
```

6.7 Programming Examples

Load the file (:l or :load) into Haskell environment and execute.

```
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :l weekDay.hs
[1 of 1] Compiling Main             ( weekDay.hs, interpreted )
Ok, one module loaded.
*Main> dayofweek 1
"Monday"
*Main> dayofweek 8
"Invalid day of week"
*Main>
```

Example: Write a Haskell snippet to sort a list.

Write the following code in a file called as sortnum.hs

```
sortnum :: (Ord a) => [a] -> [a]

sortnum [] = []

sortnum (x:xs) =

   let smallnum = sortnum [a | a <- xs, a <= x]

      bignum = sortnum [a | a <- xs, a > x]
```

```
sortnum (x:xs) =
    let smallnum = sortnum [a | a <- xs, a <= x]
        bignum = sortnum [a | a <- xs, a > x]
    in  smallnum ++ [x] ++ bignum
```

Load the file and execute to get the output.

```
*Main> :l sortnum.hs
[1 of 1] Compiling Main             ( sortnum.hs, interpreted )
Ok, one module loaded.
*Main> sortnum [2,43,123,3,53,56,8]
[2,3,8,43,53,56,123]
*Main>
```

Example: Find the largest number under 100,000 that's divisible by 1234.

```
largestDivisor :: (Integral a) => a
largestDivisor = head (filter d [100000,99999..])
```

```
largestDivisor :: (Integral a) => a

largestDivisor = head (filter d [100000,99999..])

   where d x = x `mod` 1234 == 0
```

Load the file and execute to get the output.

```
*Main> :l largestDiv.hs
[1 of 1] Compiling Main             ( largestDiv.hs, interpreted )
Ok, one module loaded.
*Main> largestDivisor
99954
*Main>
```

Example: Using recursion and the function add, define a function that multiplies two natural numbers.

```
data Nat = Zero | Succ Nat

nat2int :: Nat -> Int

nat2int Zero = 0

nat2int (Succ n) = 1 + nat2int n

instance Show Nat where
```

```
nat2int (Succ n) = 1 + nat2int n

instance Show Nat where

    show n = show (nat2int n)

add :: Nat -> Nat -> Nat

add Zero n = n

add (Succ m) n = Succ (add m n)

mul :: Nat -> Nat -> Nat


mul Zero _  = Zero

mul _ Zero = Zero

mul m (Succ n) = add m (mul m n)

-- example: 3 * 4 = 12

three = Succ (Succ (Succ Zero)) -- 3

four = Succ (Succ (Succ (Succ Zero))) -- 4
```

```
-- example: 3 * 4 = 12

three = Succ (Succ (Succ Zero)) -- 3

four = Succ (Succ (Succ (Succ Zero))) -- 4

res1 = mul three four - 12
```

Figure 49 shows you the output of this program:

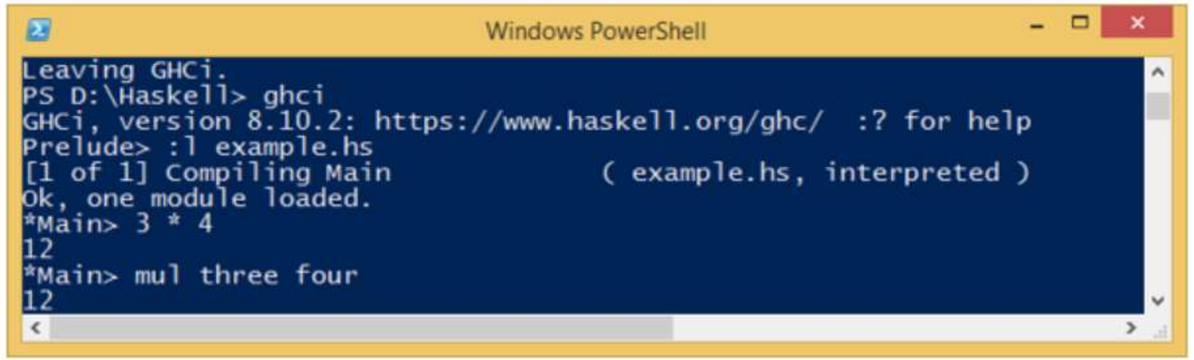


Figure 50: Output of Example

Example: Implement the game of nim in Haskell, where the rules of the game are as follows:

- The board comprises five rows of stars:

Figure 50: Output of Example

Example: Implement the game of nim in Haskell, where the rules of the game are as follows:

- The board comprises five rows of stars:
  1. *****
  2. ****
  3. ***
  4. **
  5. *
- Two players take it turn about to remove one or more stars from the end of a single row. The winner is the player who removes the last star or stars from the board.

```
-- Game of nim (without input validation)

showBoard :: [Int] -> IO ()

showBoard xs = do mapM_ putStrLn [replicate x '*' | x <- xs]
```

3. ***
4. **
5. *

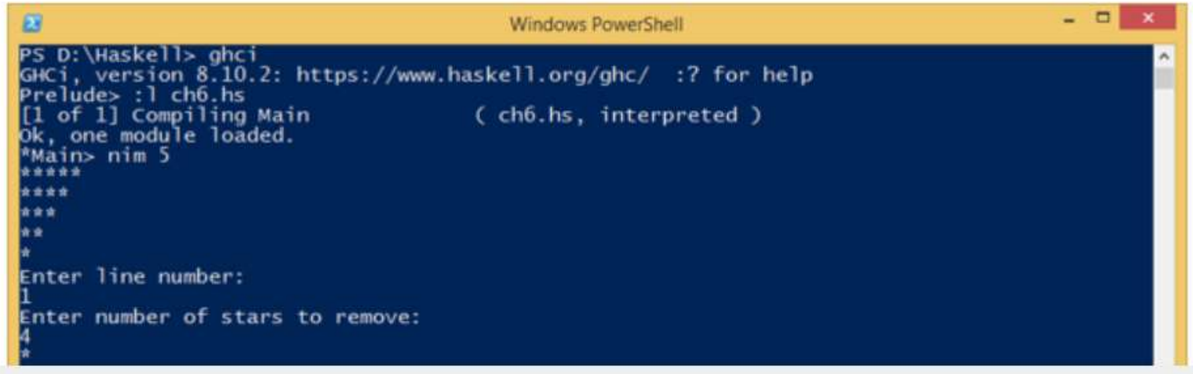
- Two players take it turn about to remove one or more stars from the end of a single row. The winner is the player who removes the last star or stars from the board.

```
-- Game of nim (without input validation)

showBoard :: [Int] -> IO ()

showBoard xs = do mapM_ putStrLn [replicate x '*' | x <- xs]

removeStars :: [Int] -> Int -> Int -> [Int]

removeStars xs l n = [if i==l then x-n else x | (i,x) <- zip [1..]
xs]

play :: [Int] -> IO ()

play board = do
```

```
removeStars :: [Int] -> Int -> Int -> [Int]

removeStars xs l n = [if i==l then x-n else x | (i,x) <- zip [1..]
xs]

play :: [Int] -> IO ()

play board = do

    showBoard board

    if sum board == 0 then

        putStrLn "finished!"

    else do

        putStrLn "Enter line number: "

        line <- getLine

        let l = (read line :: Int)

        putStrLn "Enter number of stars to remove: "

        stars <- getLine
```

```
            putStrLn "Enter number of stars to remove: "
            stars <- getLine
            let n = (read stars :: Int)
            play (removeStars board l n)
-- n :: Int == number of lines
nim :: Int -> IO ()
nim n = play (reverse [1..n])
```

Figure 50 shows you the output of this program:



SYBGEN ebooks Portal › Course
ebooks.sybgenlearning.com/appLms/index.php?modname=organization&op=custom_enditem&id_item=82


```
nim n = play (reverse [1..n])
```

Figure 50 shows you the output of this program:

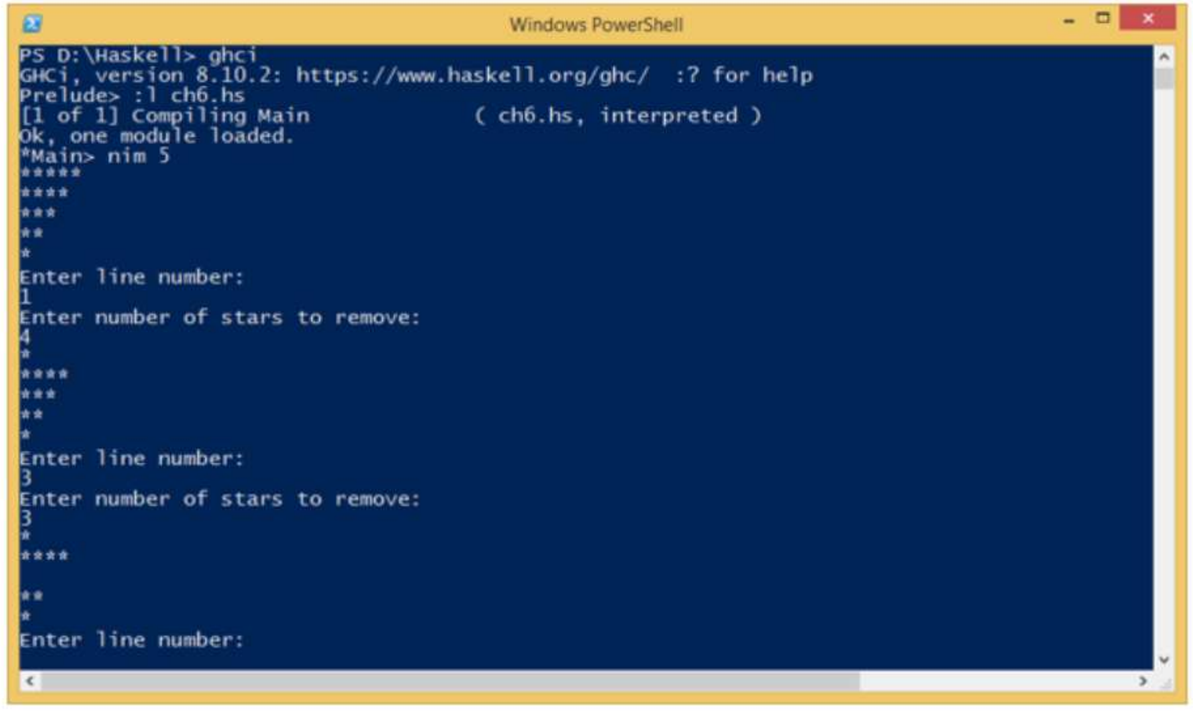


Figure 51: Output of Example

Type here to search
ENG
22:35
22-11-2020

```
Enter number of stars to remove:
3
*
****

**
*
Enter line number:
```

Figure 51: Output of Example

Example: The Fibonacci sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...starts with 0 and 1 with each further number being the sum of the previous two. Using a list comprehension, define an expression

```
fibs :: [Integer]
```

that generates this infinite sequence.

```
let fib = 0 : 1 : zipWith (+) fib (tail fib)
```

Figure 51 shows you the output of this program:

```
Windows PowerShell
Prelude>
Prelude>
Prelude>
Prelude> let fib = 0 : 1 : zipWith (+) fib (tail fib)
```

each further number being the sum of the previous two. Using a list comprehension, define an expression

```
fibs :: [Integer]
```

that generates this infinite sequence.

```
let fib = 0 : 1 : zipWith (+) fib (tail fib)
```

Figure 51 shows you the output of this program:

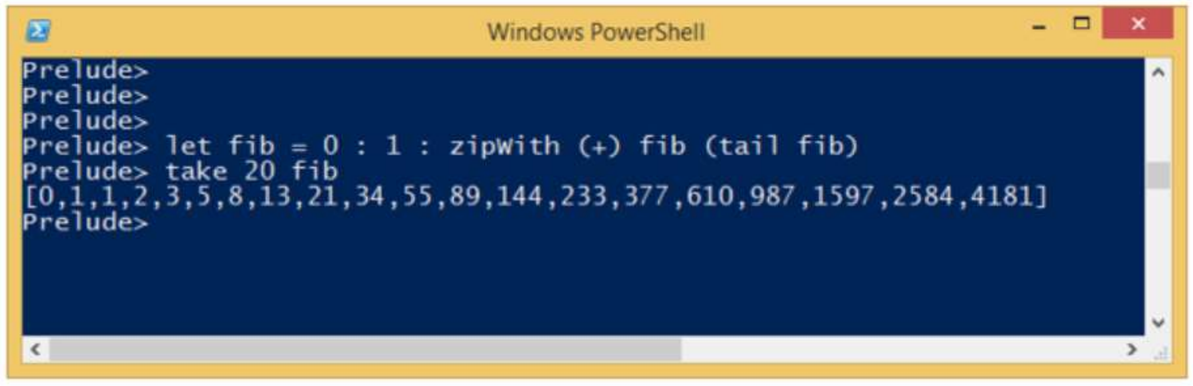


Figure 52: Output of Example

6.8 Recap and Questions

6.8 1 Recap

Let us quickly recapitulate the following discussion points:

- Haskell is a purely functional programming language.
- A functional program is a single expression, which is executed by evaluating the expression.
- Some of the common features of Haskell are statically typed, concurrent, purely functional, type inference and lazy.
- Haskell is able to decode any kind of number as a number. It is not necessary to mention its type.
- Haskell can also identify a character given in as an input to it. A single character is enclosed in single quotes.
- A type system in Haskell provides an abstraction at the lowest level. It helps us to prevent accidently mixing those types.
- A type system in Haskell provides an abstraction at the lowest level. It helps us to prevent accidently mixing those types.
- Pattern matching means to check if some data matches a certain pattern or a type of expression and then acts accordingly.
- Guards helps in performing fixed tests of a value's shape. They help in testing whether a value has a certain property.
- Anonymous functions are known as "lambda" functions

SYBGEN ebooks Portal › Course
ebooks.sybgenlearning.com/appLms/index.php?modname=organization&op=custom_enditem&id_item=82

- Haskell is a purely functional programming language.
- A functional program is a single expression, which is executed by evaluating the expression.
- Some of the common features of Haskell are statically typed, concurrent, purely functional, type inference and lazy.
- Haskell is able to decode any kind of number as a number. It is not necessary to mention its type.
- Haskell can also identify a character given in as an input to it. A single character is enclosed in single quotes.
- A type system in Haskell provides an abstraction at the lowest level. It helps us to prevent accidently mixing those types.
- A type system in Haskell provides an abstraction at the lowest level. It helps us to prevent accidently mixing those types.
- Pattern matching means to check if some data matches a certain pattern or a type of expression and then acts accordingly.
- Guards helps in performing fixed tests of a value's shape. They help in testing whether a value has a certain property.
- Anonymous functions are known as "lambda" functions.
- A higher-order function is defined as a function that takes other functions as arguments or returns a function as result.
- A Haskell module is a collection of related functions, types and typeclasses.
- A monad is defined as a way for values that can be used in sequences of computations, resulting in a structure of computations.
- Monoid typeclass is a class that have an operation for combining values with a value that does not do anything when combined with others.

6.8.2 Test Your Skills- Multiple Choice Questions

Type here to search
ENG
22:36
22-11-2020

SYBGEN ebooks Portal › Course
ebooks.sybgenlearning.com/appLms/index.php?modname=organization&op=custom_enditem&id_item=82

- A type system in Haskell provides an abstraction at the lowest level. It helps us to prevent accidently mixing those types.
- Pattern matching means to check if some data matches a certain pattern or a type of expression and then acts accordingly.
- Guards helps in performing fixed tests of a value's shape. They help in testing whether a value has a certain property.
- Anonymous functions are known as "lambda" functions.
- A higher-order function is defined as a function that takes other functions as arguments or returns a function as result.
- A Haskell module is a collection of related functions, types and typeclasses.
- A monad is defined as a way for values that can be used in sequences of computations, resulting in a structure of computations.
- Monoid typeclass is a class that have an operation for combining values with a value that does not do anything when combined with others.

6.8.2 Test Your Skills- Multiple Choice Questions

1. Which of the following properties of Haskell ensures that when a program is compiled, the compiler will know which piece of code is a string?

a. Statically typed
b. Purely functional
c. Concurrent
d. Lazy

Type here to search ENG 22:36 22-11-2020

- Monoid typeclass is a class that have an operation for combining values with a value that does not do anything when combined with others.

6.8.2 Test Your Skills- Multiple Choice Questions

1. Which of the following properties of Haskell ensures that when a program is compiled, the compiler will know which piece of code is a string?

a. Statically typed
b. Purely functional
c. Concurrent
d. Lazy

2. Which of the following properties of Haskell ensures Haskell does not evaluate any express without a reason?

a. Statically typed
b. Purely functional
c. Concurrent
d. Lazy

3. Which of the following is a compiler used to compile Haskell programs?

6.8 Recap and Questions

2. Which of the following properties of Haskell ensures Haskell does not evaluate any express without a reason?

a. Statically typed
b. Purely functional
c. Concurrent
d. Lazy

3. Which of the following is a compiler used to compile Haskell programs?

a. Open64
b. GHC
c. GCC
d. MinGW

4. Which of the following is used to identify String in Haskell?

a. "Hello"
b. Hello
c. 'Hello'
d. `Hello`

6.8 Recap and Questions

3. Which of the following is a compiler used to compile Haskell programs?

a. Open64
b. GHC
c. GCC
d. MinGW

4. Which of the following is used to identify String in Haskell?

a. "Hello"
b. Hello
c. 'Hello'
d. `Hello`

5. Which of the following is classified as a list?

a. (a, b, c)
b. [a, b, c]
c. [1, 2, 3]
d. ['1', '2', '3']

SYBGEN ebooks Portal › Course
ebooks.sybgenlearning.com/appLms/index.php?modname=organization&op=custom_enditem&id_item=82

5. Which of the following is classified as a list?

a. (a, b, c)
b. [a, b, c]
c. [1, 2, 3]
d. ['1', '2', '3']

6.8.3 Fill in the Blanks

1. _____ command is used to find the type of the expressions.
2. _____ function returns the first element of a list.
3. _____ typeclass is used for the types that can test for equality.
4. _____ typeclass is used for the types that provide the functionality of ordering. Its members implement >, <, >=, <= and compare functions.
5. _____ typeclass provides the sequential or ordered functionality. Its advantage is that its types can be used in list ranges.

6.8.4 Theoretical Questions

1. Explain functional programming and its goals.

Type here to search ENG 22:36 22-11-2020

d. ['1', '2', '3']

6.8.3 Fill in the Blanks

1. _____ command is used to find the type of the expressions.
2. _____ function returns the first element of a list.
3. _____ typeclass is used for the types that can test for equality.
4. _____ typeclass is used for the types that provide the functionality of ordering. Its members implement >, <, >=, <= and compare functions.
5. _____ typeclass provides the sequential or ordered functionality. Its advantage is that its types can be used in list ranges.

6.8.4 Theoretical Questions

1. Explain functional programming and its goals.
2. Describe the features of Haskell.
3. Explain the basic data types in Haskell.
4. Describe the types and typeclasses in Haskell.
5. Discuss the functions in Haskell.

Chapter 7 : Logic Programming

Learning Objectives

At the end of this module, you will be able to:

- Explain logic programming with Prolog
- Discuss resolution and unification
- Describe lists
- Describe arithmetic and search operations
- Explain control flow and database manipulation

7.1 Introduction

Logic is majorly used in the design of digital circuit and most programming languages provide a logical (Boolean) type and operators. It is also used in formal study of language semantics. Logic programming allows the programmer to define a collection of axioms with the help of which theorems can be proven. The user of a logic program defines a theorem and the language attempts to find a collection of axioms and inference steps that helps in proving the theorem. Of the several existing logic languages, Prolog is by far the most widely used.

In this chapter, you will learn about the logic programming with Prolog. You will learn resolution and unification. You will learn about lists, arithmetic and search operations. At the end of this chapter you will learn about control flow and database manipulation.

7.2 Logic Programming with Prolog

Logic Programming is different from traditional programming languages such as C, C++ and Java. It is a remarkable programming language. Computer scientists are using logic programming to allow machines for a reason because it is capable of knowledge representation. In logic programming, logic represents the knowledge and inference which is used to manipulate programs. However, the logic which is used in logic programming is a subset of first-order predicate logic and which will be in a clausal form. The first first-order predicate logic is used because it can be understood and capable to represent all computational problems. Where, resolution inference system is used to manipulate the knowledge which will be needed for theorem- proving in clausal-form logic as shown in Figure 1:

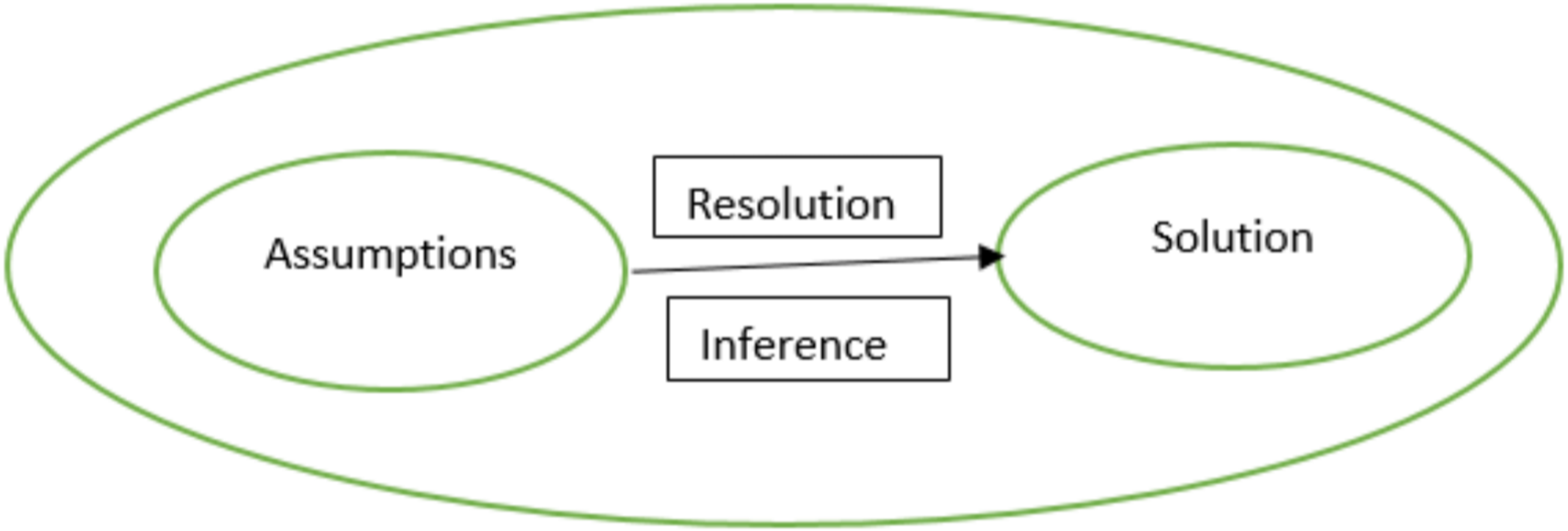


Figure 1: Clausal-Form Logic

Figure 1: Clausal-Form Logic

Prolog programming language is based on logic programming. In 1972, Prolog was developed and implemented in Marseille, France by Alain Colmerauer with Philippe Roussel, based on Robert Kowalski's procedural interpretation of Horn clauses. The main idea behind Prolog programming language was to make a program look like a logic. It is a logical and declarative programming language which means that program data comprise based on the facts and rules rather than computing i.e. how to solve it.

Prolog was one of the first logic programming languages and today still remains the most popular such language because of its several free and commercial implementations available such as theorem proving, expert systems, term rewriting, type systems, and automated planning and it's also intended use of natural language processing field.

Advanced Prolog environments support the graphical user interfaces, administrative and networked applications. It is also well suited for specific tasks which are obtained from rule-based logical queries, for example, searching databases, voice control systems, and filling templates. However, a basic logic programming environment has no literal values. Where an identifier with upper case letters and other identifiers signify variables and identifiers with lower-case letters signify data values. Thus, the basic Prolog elements are typeless and implementations include integer value, characters, and operations and tuples and lists describe the mechanism of Prolog. To obtain the conclusion, the user simply asks a question instead of running a program. When a user asks a question, then the run time system searches the particular request through the database of facts and rules to determine the conclusion. Thus, to decrease the answer, there is more than one way. In this case, the run time system will be asked to find another solution. To generate another solution, use the backtracking strategy. Prolog is a weakly typed language with static scope rules and dynamic type checking.

Prolog is the fast-incremental development cycle and rapid prototyping capabilities have encouraged the use of the language as a tool for solving artificial intelligence problems. It is a powerful tool for building robust commercials applications. Prolog features include interfaces to other languages and data products, stand-alone applications generators and recently, it has started supporting techniques such as orientated and constraints-based programming.

Constraints which replaces Prolog usually are pattern matching mechanism with a more general operation known as constraints satisfaction. Constraints are a way to reduce the size of the search space and thus increase the efficiency of the scheduler. Some common Prolog applications are specification language, robot planning, natural language understanding, machine learning, problem-solving, intelligent database retrieval, expert system and automated reasoning.

7.2.1 Types of Prolog

Prolog is used to provide tuples, lists, numbers, atoms, and patterns. Here, the type of objects which are passed as arguments.

1. **Simple:** This type of Prolog is implementation dependent. The table 1 shows the implementation of a simple type of Prolog is as follows:

7.2.1 Types of Prolog

Prolog is used to provide tuples, lists, numbers, atoms, and patterns. Here, the type of objects which are passed as arguments.

1. **Simple:** This type of Prolog is implementation dependent. The table 1 shows the implementation of a simple type of Prolog is as follows:

Table 1: Simple Type

Type	Output
Boolean	True, Fail
Variables	Variable
Integer	Integer
Atom	Character sequence
Real	Floating-point sequence

- **Boolean:** Boolean is a subset of algebra used for creating true/false statements. It uses operators such as AND, OR, XOR, and NOT to compare values and return a true or false output.
- **Variables:** A variable refers to a symbol which represents a value like "a" or "b,". In algebraic equations, the value of one variable is generally dependent on the value of another.
- **Integer:** An integer refers to the whole number which can be positive, negative, or zero. For example, 9, -25, and 5,148.
- **Atom:** Atoms are constants which have no numerical value. All the atoms start with a lower-case letter or small letter.
- **Real:** A real number is any positive or negative number. It includes all integers and all rational and irrational numbers. It also includes decimal points which are also known as a floating-point number.

- **Composition**: It distinguishes between data and program which are blurred and captured in Prolog. In a program, an argument is often passed to predicates the data. Thus, in Prolog, the most common data structure used is listed. Lists are much similar to the array and stack in which it is helpful to access the lists of elements sequentially. Prolog is used to allow arbitrary patterns like data and which can represent tuples pattern. Thus, an array is not provided by Prolog but single or multidimensional arrays can be represented as a list or list of lists. It can also be represented by a set of facts in the database. The syntax is as follows:

TYPE REPRESENTATION [comma separated sequence of items] list pattern sequence of items

Using the square brackets ([]+), a prolog list can represent.

In this example, the list shows the elements of the book Expert systems, Java, Machine learning.In prolog list, the elements are ordered even if there are no indexes are present, still, the elements will also be in order.

[Expert systems, Java, Machine learning]

- **Predicates**: In this type, the user needs to define the parameter type because Prolog is a weakly typed language. To define the parameter type in Prolog, Table 2 consists of built-in predicate which can be used:

Table 2: Predicates Type

Type	Output
atom(A)	A is an atom
atomic(A)	A is a number of an atom

Table 2: Predicates Type

Type	Output
atom(A)	A is an atom
atomic(A)	A is a number of an atom
number(N)	N is an integer or real value
var(V)	V is a variable
non var(NV)	NV is not a variable
integer(I)	I am an Integer
real(R)	R is a floating-point number
T =L	T is a term, L is a list

T =L	T is a term, L is a list
functor(T,F,A)	T is a term with functor F, and A is an arity
clause(H, T)	H:- T is a program rule

7.2.2 Data objects

In Prolog, the data objects are known as terms. Atoms and compound terms have a special importance in Prolog clauses and are known call terms. In terms, some examples which have been used in Prolog programs are Type, (henry), X and cat(X).There are some common types of term are as follows:

- **Numbers:** In Prolog all versions allow the use of integers. Where an integer is a whole number can be positive, negative, or zero. It optionally preceded by a + or - sign, for example: 500, -56, +2 and 022.

In modern versions of Prolog it also allows the use of numbers with decimal points which are written in a similar way as integers, but it consists of a single decimal point and an optional positive, negative, or zero signs like 5.40 -.350 +150.

- **Atoms:** Atoms are constants which do not have a numerical value. All the atoms start with a lower-case letter or small letter**.** There are three methods in which atoms can be written are as follows:
    - A sequence which consists of one or more letters and it is written in upper or lower case, numerals and underscores, beginning with a lower-case letter. For example:

today_is_Wednesday

rainy_day

in_everywhere

- A sequence which consists of one or more letters and it is written in upper or lower case, numerals and underscores, beginning with a lower-case letter. For example:

today_is_Wednesday

rainy_day

in_everywhere

but not

Today

today-is-Wednesday

ineverywhere

- A sequence consists of characters which are enclosed in single quotes, including spaces and upper-case letters. For example,

'Today is Wednesday'

'today-is-Wednesday'

'ineverywhere'

- A sequence consists of one or more special characters in a list which contains the following + - * / > < = & # @ symbols. For examples:

+++

- A sequence consists of one or more special characters in a list which contains the following + - * / > < = & # @ symbols. For examples:

  +++

  >=

  >

  +--

- **Variables:** A variable refers to a symbol which represents a value like "a" or "b,". In algebraic equations, the value of one variable is generally dependent on the value of another. The name of a variable can be defined by a sequence of one or more letters with upper or lower case, numerals underscores and beginning with an upper-case letter or underscore. For example,

Author name

Book_B

_120B

but not

120_B

Book-B

Author name

Note: *The variable which consists of a single underscore is called as the anonymous variable and it is reserved for a special purpose.*

- **Compound Terms:** In the prologue, the compound terms are important fundamental of programs. It is a structured data type which starts with an atom, called as a functor. The functor is written in a sequence of

- **Compound Terms:** In the prologue, the compound terms are important fundamental of programs. It is a structured data type which starts with an atom, called as a functor. The functor is written in a sequence of one or more arguments, which are enclosed by brackets and separated by commas.

  Syntax:

  functor(t1,t2, ... ,tn) n≥1

Just like other programming languages, a compound term is helpful in representing a record structure. An example of compound terms are as follows:

like(john,Logicprolog)

write(X)

read(john)

rewrite(X)

>(1,2)

person('Amy smith',10,engineer,Boston)

Here, the functor represents the record of the name, where the arguments represent the field records. The number of arguments which are passed as a compound term is known as called its arity. t of a compound term should be a term, which can be of any kind including a compound term.

1. **Lists:** A list is considered as a special type of compound term. The Lists are written as an unlimited and consist of a number of arguments which generally known as list elements. It is enclos brackets and separated by commas. For example, [A,B,C,]. A list with no elements is known as the empty list. It is written as [].

When a program is written in an advanced it does not decide a list of the number of elements will be the arity of a compound term. At this stage, it is necessary to aware of that a list of element can be a term of any kind and also including a compound term list. For example,

[john,leo,y,myfriend(a,b,c),[x,y,z],w]

[[smith,18],[amy,50,teacher],joey,parent(marco,hugo),[a,b,[c,d,e],f],35]

[john,leo,y,myfriend(a,b,c),[x,y,z],w]

[[smith,18],[amy,50,teacher],joey,parent(marco,hugo),[a,b,[c,d,e],f],35]

[[ports,houston,US,dover],[ports,US, houston],[South Louisiana]]

7.2.3 Installation of GNU Prolog

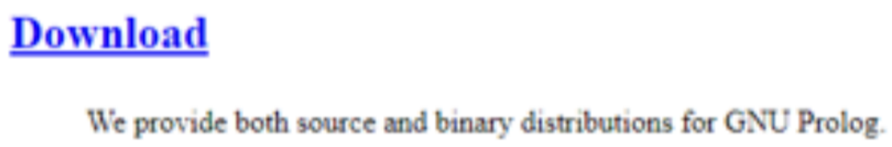
GNU Prolog is a free source compiler with constraint solving and it was developed by Daniel Diaz. It accepts programs and gives output in native binaries code such as GCC does from a C source. The obtained executable output is stand-alone. The size of the executable output can be very small because since it avoids to link the code which is unused inbuilt predicates.

The performances of GNU Prolog compiler is very encouraging as compared to commercial systems because, besides the native-code compilation, it provides a classical interactive interpreter for top-level with a debugger. The ISO standard for Prolog is attached with many extensions which are useful in practice for example global variables, OS interface, sockets. It includes efficient Finite Domains (FD) over constraint solver to run a program. It also opens constraint logic programming solver to the user by combining the power of constraint programming to the declarative logic programming.

The GNU Prolog compiler is based on the Warren Abstract Machine (WAM). First, it compiles a program to a WAM file then translate to a low-level machine-independent language which generally known as mini-assembly. It is specifically designed for GNU Prolog to get an object which has been predicted as the result. It also allows complier to give a native stand-alone output. The main advantage of the compiler is its produces native code fast. The feature is that output size will be very small and the code of most unused inbuilt predicates also may be excluded from the output during link-time.

Let us learn how to install the GNU prolog compiler environment by following steps:

1. Go the http://www.gprolog.org/ to download the GNU prolog complier. The binary distributions will provides the environment compatibility to download the version as shown in Figure 2:



Download

We provide both source and binary distributions for GNU Prolog.

Source distributions:

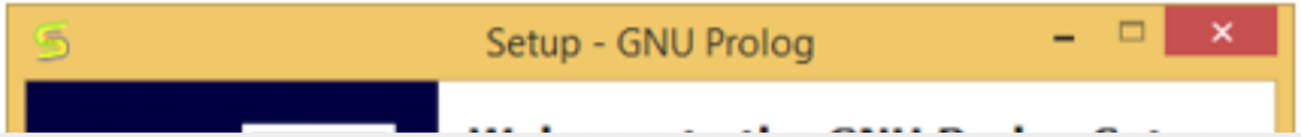
- the main source distribution gprolog-1.4.5.tar.gz.

Binary distributions:

- Mac OS X High Sierra installer package (created using MacPorts by Paulo Moura: installs GNU Prolog in /opt/local/ and /opt/local/bin).
- Windows intel 32 bits auto-install setup (compiled under ix86 / Windows 7 with MSVC++).
- Windows intel 32 bits auto-install setup (compiled under ix86 / Windows 7 with MinGW gcc under MSys).
- Windows intel 64 bits auto-install setup (compiled under x86_64 / Windows 7 with MSVC++).
- Windows intel 64 bits auto-install setup (compiled under x86_64 / Windows 7 with MinGW64 gcc under Cygwin).

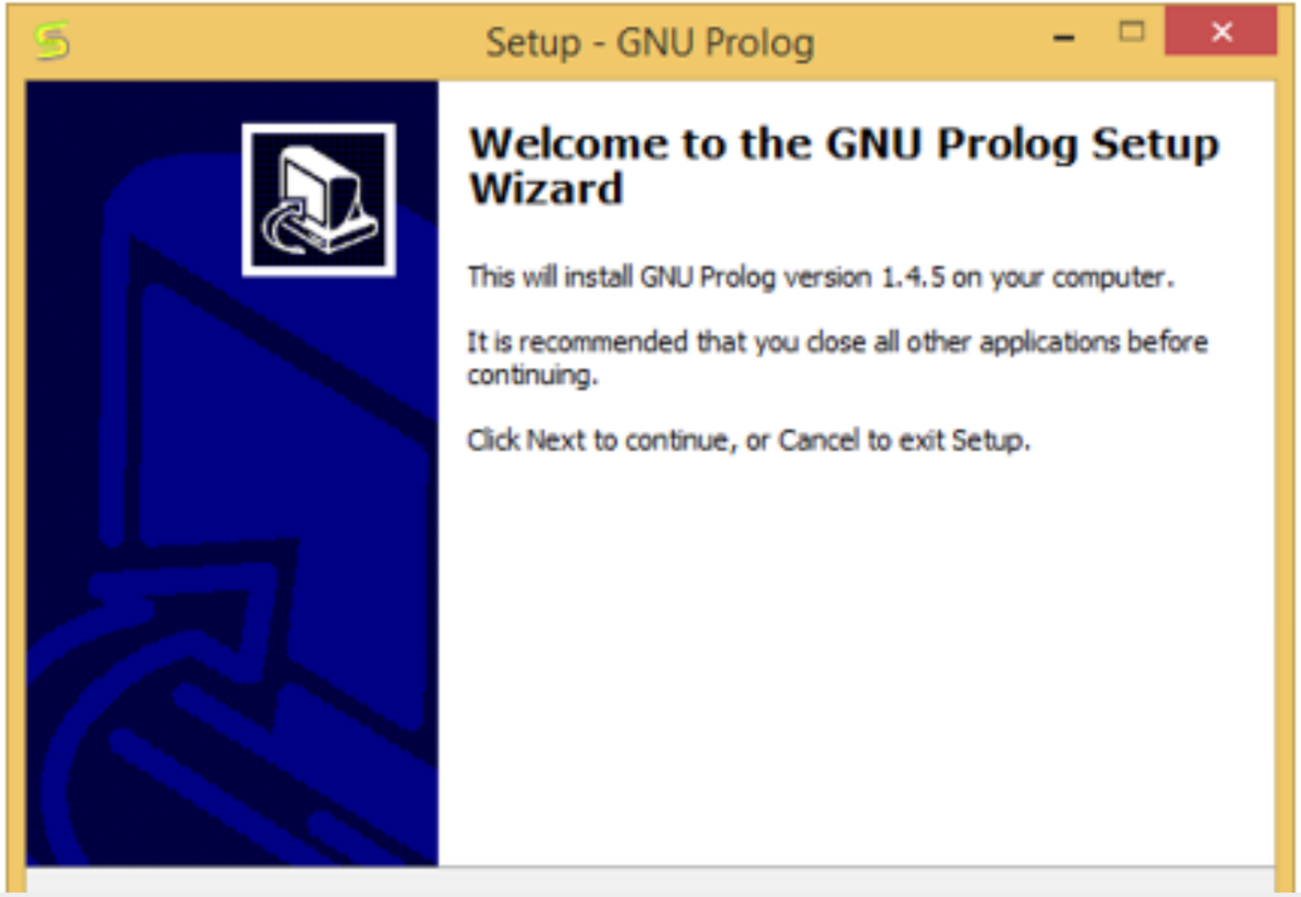
Figure 2: Download Version

2. Select the environment compatibility and download.
3. Click download GNU prolog complier, you can run the complier to install the version 1.4.5 as shown in Figure 3:



2. Select the environment compatibility and download.

3. Click download GNU prolog complier, you can run the complier to install the version 1.4.5 as shown in Figure 3:



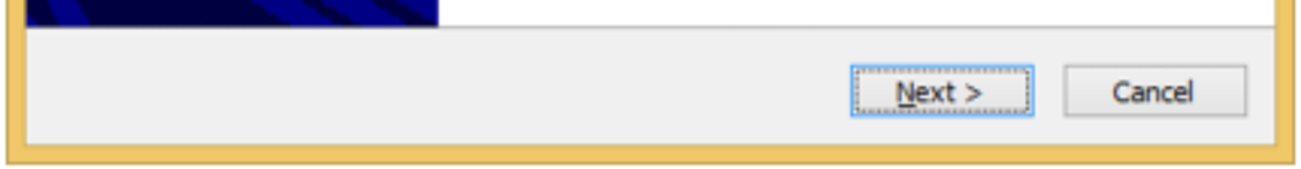
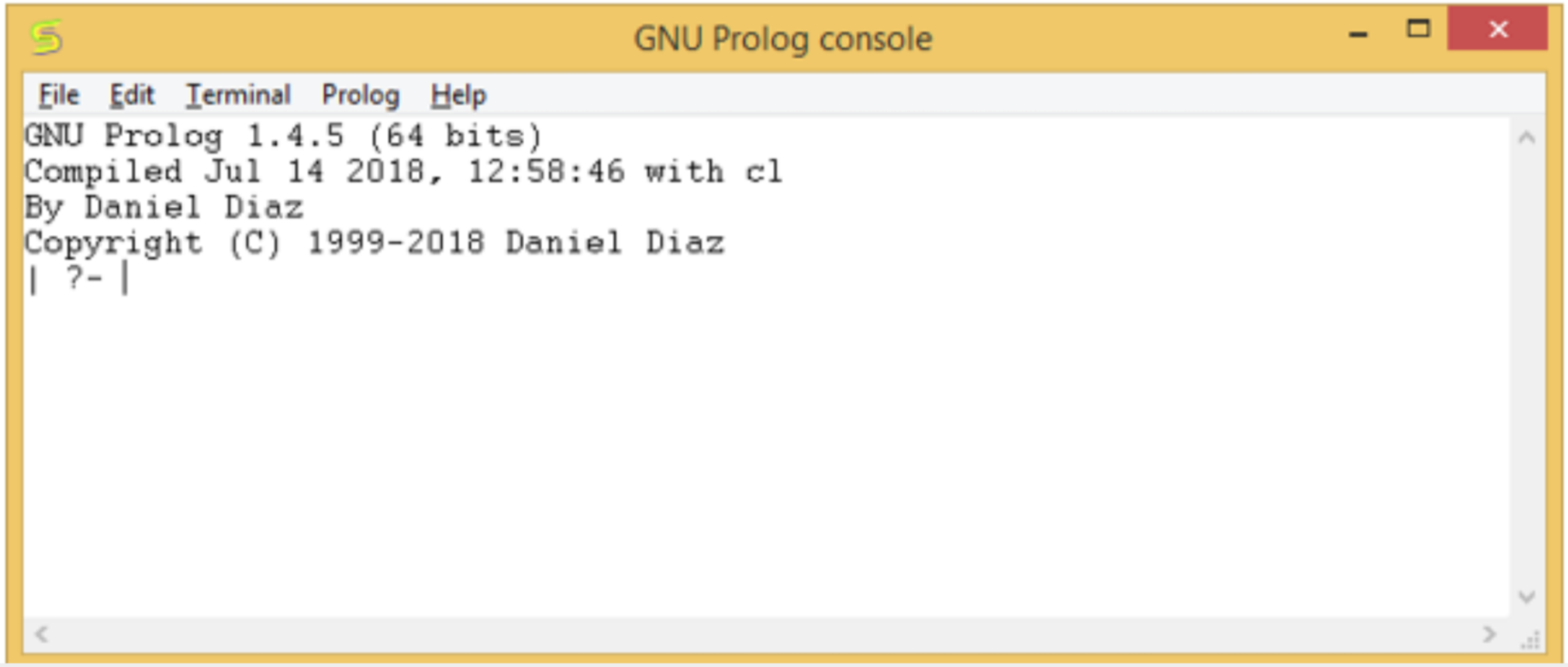


Figure 3: Downloading

After successful installation of the GNU prolog complier a GNU prolog console window will appear as shown in Figure 4:



7.3 Resolution and Unification

Prolog is a simple programming language. It consists of a single data structure that is referred to expression trees and a single operation on those data structures that is known as unification. There is one inference rule (resolution) and no axioms. Despite this, Prolog is an extremely powerful programming language. It can be used to implement Turing Machines, or theorem provers.

7.3.1 Resolution Rule for Propositional Logic Programs

Prolog programs are written in a special format that is known as clausal form. Formally a clause is defined as an expression of the form:

$$A_1,\ldots,A_m \neg B_1,\ldots,B_n$$

where each of the As and Bs stand for propositions.

A clause is used to define to express a logical implication of the following sort:

- if all of the Bs are true, then one of the As must be true as well.

This rule, which provides a semantics for clauses, is also true of the expression

$$B_1 \grave{U} \ldots \grave{U}\ B_n \ ®\ A_1 \acute{U} \ldots \acute{U}\ A_m$$

These two expressions are equivalent, it means that they mean the same thing. Another method used to put the same rule of interpretation:

- either at least one of the Bs is false or at least one of the As is true.

Another way of representing them is:

$$\{A_1, \ldots ,A_m,\neg B_1, \ldots , \neg B_n \}$$

$$\{A_1, \ldots, A_m, \neg B_1, \ldots, \neg B_n\}$$

The elements of such sets are known as literals and the set is interpreted disjunctively. Given this interpretation, the clause

$A_1, \ldots, A_m \neg$

must mean that the disjunction of the As (A_1Ú...Ú A_m) is true, since it is impossible for one of the *Bs* to be false. Also, the clause

$\neg B_1, \ldots, B_n$

means that the conjunction of the Bs (B_1Ù...Ù B_n) is false, since it is impossible for one of the As to be true.

Resolution in Prolog is basically the inference mechanism. Suppose there are two clauses m :- b and t :- p, m, z. So, from that you can infer t :- p, b, z. This is known as resolution. It means that when two clauses are resolved, a new clause is created. The resolution principle says that if C1 and C2 are Horn clauses and the head of C1 matches with one of the bodies of C2, then you can replace the term in C2 with the body of C1. For example,

```
takes(ronald_jones, xyz201).

takes(ronald_jones, abc301).

takes(wes_king, gef401).

takes(wes_king, abc301).

batchmate(X, Y) :- takes(X, Z), takes(Y, Z).
```

If X is ronald_jones and Z be abc301, then you can replace the first term on the right-hand side of the last clause with the (empty) body of the second clause. It creates a new rule

```
classmates(ronald_jones, Y) :- takes(Y, abc301).
```

In other words, Y is a classmate of ronald_jones if Y takes abc301. For a variable (Z) on the right-hand side that does not appear in the head. Such variables are existentially quantified. It means that for all X and Y, X and Y are classmates if there exists a class Z that they both take.

7.3.2 Unification

Unification in Prolog is possible if two expressions can be made to have “the same structure” through variable instantiation. It means that an instantiated variable will not change its value. However, it can become un-instantiated when backtracking occurs. Some of the examples are as follows:

1. parent(X,Y) and parent(albert, edward)

These unify: X=albert, Y=edward

2. parent(X,edward) and parent(albert,Y)

These unify: X=albert, Y=edward

3. parent(albert,edward) and parent(victoria,Y)

These do not unify, since albert and victoria do not unify.

4. parent(X, edward) and parent(Y, edward)

These unify: X=Y (or Y=X or creating a variable Z and letting X=Z and Y=Z).

5. parent(X,Y) and female(X)

These do not unify, since parent and female do not unify.

6. parent(albert,edward) and parent(X,Y,Z)

These do not unify, since the first expression has two arguments and the second expression has three arguments

In order for two expressions to unify, they must have the same functor and the same number of arguments or else there is no way for them to have the same structure. In Prolog, = is used for unification.

In order for two expressions to unify, they must have the same functor and the same number of arguments or else there is no way for them to have the same structure. In Prolog, = is used for unification.

GNU Prolog console

File Edit Terminal Prolog Help

```
GNU Prolog 1.4.5 (64 bits)
Compiled Jul 14 2018, 12:58:46 with cl
By Daniel Diaz
Copyright (C) 1999-2018 Daniel Diaz
| ?- parent(X,Y) = parent(albert,Z).

X = albert
Z = Y

yes
| ?- parent(X,Y) = parent(A,B).

X = A
Y = B

yes
| ?- parent(X,Y) = course(A,B).

no
| ?- parent(X,X) = parent(albert,Y).

X = albert
Y = albert
```

```
X = albert
Y = albert

yes
| ?- parent(X,Y,Z) = parent(A,B)
```

Figure 5: Examples of Unification

Unification is the process where the variables are instantiated, or allocated memory and assigned values, so that patterns match during resolution. It is also used to make the two terms same. Equality is the basic expression that is determined by unification.

In Prolog, the goal s = t attempts to unify the terms s and t. It succeeds if unification succeeds and fails otherwise. Therefore, you can learn unification in Prolog by experimenting with the effect of equality, as shown in Figure 6:

GNU Prolog console

File Edit Terminal Prolog Help

```
GNU Prolog 1.4.5 (64 bits)
Compiled Jul 14 2018, 12:58:46 with cl
By Daniel Diaz
Copyright (C) 1999-2018 Daniel Diaz
| ?- me = me.

yes
```

```
yes
| ?- me = you.

no
| ?- me = X.

X = me

yes
| ?- f(a, X) = f(Y, b).

X = b
Y = a

yes
| ?- f(X) = g(X).

no
| ?- f(X) = f(a, b).

no
| ?- f(a, g(X)) = f(Y, b).

no
| ?- f(a, g(X)) = f(Y, g(b)).
```

```
no
| ?- f(a, g(X)) = f(Y, b).

no
| ?- f(a, g(X)) = f(Y, g(b)).

X = b
Y = a

yes
| ?- |
```

Figure 6: Experiments with the Equality

From these experiments, you can formulate the following unification algorithm for Prolog:

1. A constant unifies only with itself: me = me succeeds but me = you fails.
2. A variable that is uninstantiated unifies with anything and becomes instantiated to that thing.
3. A structured term unifies with another term only if it has the same function name and the same number of arguments, and the arguments can be unified recursively. Thus, f(a, X) unifies with f(Y, b) by instantiating X to b and Y to a.

7.4 Lists

In Prolog, a list refers to an ordered sequence of elements. Lists are a common data structure in computer programming. It is represented in tree form consists of structures with arguments such as an empty list or two arguments i.e. head and a tail. However, in the list the head argument stores a particular term and where the end of a list is represented as a tail which is set to an empty list.

In Prolog functor (name) of "." is the data structure used to represent lists. Thus, a single element in a list can be represented as [a] and other elements a, b and c in a list can be represented as [a, b, c]. An empty list can be represented as [].

List notation consists of elements of the list separated by commas, and the whole list enclosed in square brackets. When a program is written in an advanced it does not decide a list of the number of elements will be the arity of a compound term. At this stage, it is necessary to aware that a list of the element can be a term of any kind and also including a compound term list.

For example,

[john,leo,y,myfriend(a,b,c),[x,y,z],w]

[[smith,18],[amy,50,teacher],joey,parent(marco,hugo),[a,b,[c,d,e],f],35]

[[ports,houston,US,dover],[ports,US, houston],[South Louisiana]]

In Prolog the | character syntax is used to separate the head and tail of a list. Thus, the Prolog syntax [a|b] represents a list with head a and tail b.

7.4.1 Represent List as data

In Prolog, compound terms offer a flexible way to represent data, especially when an arguments can be consider in other terms of any complexity. For example, mypred(b,-1.3,pred2(p,pred3(3,q,bbb),r)). However, compound terms have a limitation that is each predicate should have a fixed number of arguments. For more than two arguments mypred command is used but in Prolog these have two entirely different predicates.

To overcome this problem, Prolog provides a flexible data object known as a list. A list is written in ordered sequence of elements which is called list elements and separated by commas and enclosed in square brackets. A list element does not contain an atom. It can be any Prolog term, like bound or unbound variable or another list. Thus, [X,Y,mypred(a,b,c),[p,q,r],z] is a valid list. A list element which is itself a list is known as a sublist.

7.4.2 Notations

According to the list definition, lists elements have been written as an ordered sequence and separated by commas and enclosed in square brackets it is known as 'standard bracketed notation'.

The elements consist of a sequence of one or more list elements. Where, the list part represents a list. The list [elements | list] represent an augmented version of the list with an ordered sequence of elements. Although, lists are not written in this notation but in queries a user entered the values at the system prompt.

Example 1 shows a list entered in the notation in queries entered by the user at the system prompt:

Example 1: Entering a List

?- a=alpha,b=14,c=[alpha,beta],write('List is: '),write([a,b,c]),nl.

Figure 7 shows the output

```
yes
| ?- |
```

Figure 7: Entering a List

In example 1, a query is entered by a user at the system prompt. Where, three elements have been taken X, Y and Z with alpha, 27 , [alpha, beta] values . In this notation a list is shows more than two parts which are joined together by the vertical bar character | which is known as the cons character. Thus, a list is represented by the notation [elements | list].

7.4.3 Decomposing

On every elements of a list required common operation to perform the same operation. It can be done by processing a list through its elements one by one from left to right. Thus, it can be achieved by separating the list into its head and tail in a recursive and using the list constructor a list can be separated into its head and tail.

The predicate “writeall” refers to writes out the elements of a list in per line. The “writeall” refers to many user-defined predicates for list processing. The first clause of “write all” split a list into its head A and tail L then writes out A and a newline, thus calls itself again. The second clause of “writeall” makes sure that each evaluation terminates when there are no further elements present in the list for output. The syntax is as follows:

```
/* write out the elements of a list, one per line */

writeall([]).

writeall([A|L]):- write(A),nl,writeall(L).
```

7.4.4 List operations

The ability to represent data in the form of lists is such a valuable feature of Prolog that several built-in predicates have been provided for it. The most commonly used of these are described in this and the following sections.

7.4.5 Member

The member operation accepts two arguments. The first argument can be any term except a variable and the second argument must be a list. It succeeds when the first argument is a member of the list denoted by the second argument. The syntax is as follows:

?- member (H, T).

Example 2 shows member operation in Prolog:

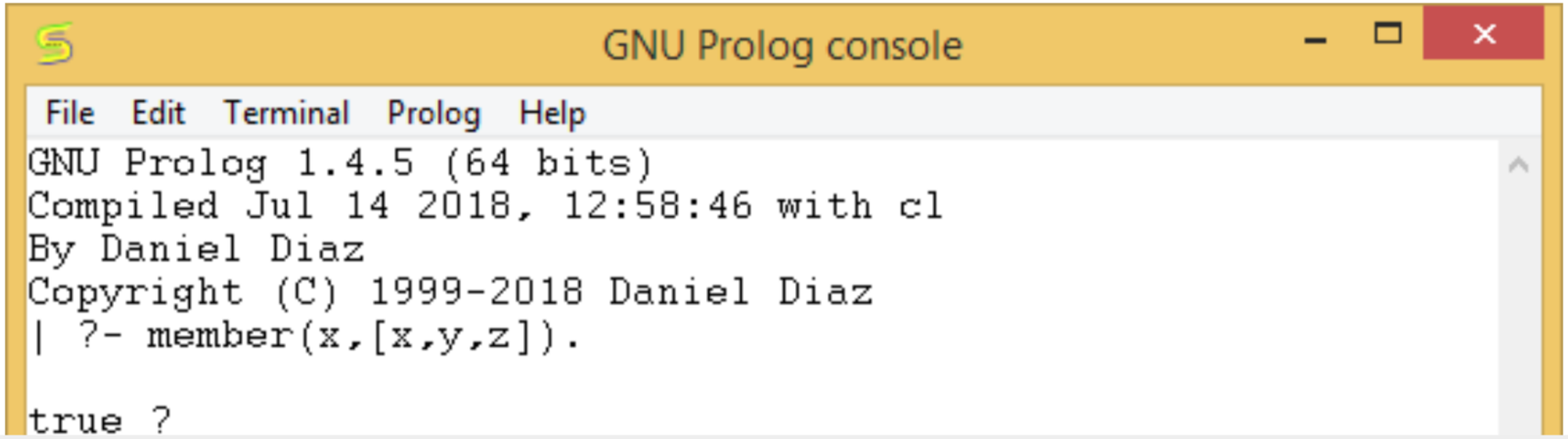
Example 2: Illustrate Member Operation

?- member(x,[x,y,z]).

?- member(mypred(x,y,z),[q,r,s,mypred(x,y,z),w]).

?- member(v,[]).

Figure 8 shows the output of Example 2:



```
yes
| ?- member(mypred(x,y,z),[q,r,s,mypred(x,y,z),w]).

true ?

yes
| ?- member(v,[]).

no
| ?-
```

Figure 8: Output of Example 2

7.4.6 Length

Length defines the size of an object or distance from one point to another. It measure the distance between two points. It accepts two arguments. The first argument is a list, where the second is an unbound variable and it is bound to the length of the list. The syntax is as follows:

?-length(T, X).

Example 3 shows length of the elements:

Example 3: Evaluate Length

?- length([a,b],M).

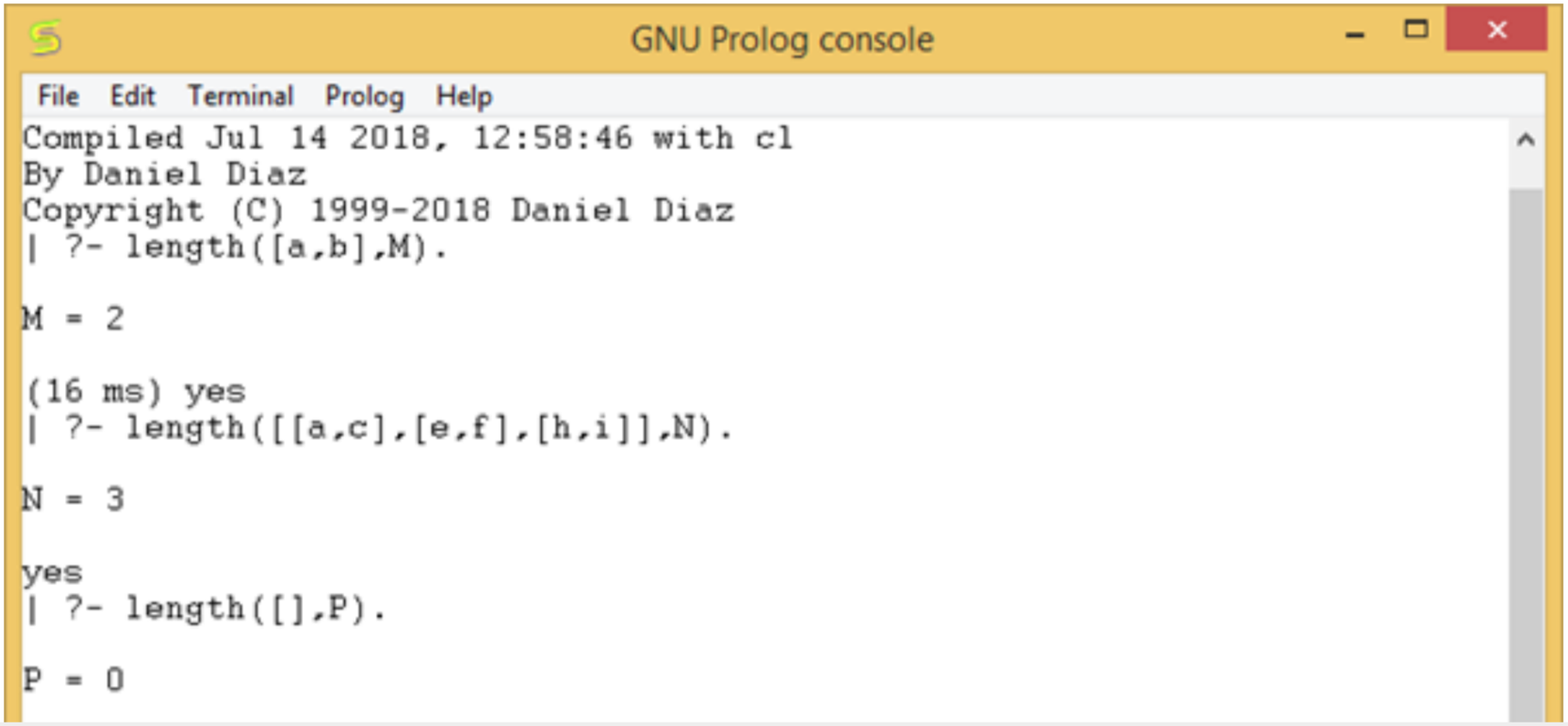
?- length([a,b],M).

?- length([[a,c],[e,f],[h,i]],N).

?- length([],P).

?- length([a,b,c],3).

Figure 9 shows the output of Example 3:



```
yes
| ?- length([a,b,c],3).

yes
| ?-
```

Figure 9: Use of Length Operation

7.4.7 Reverse

Reverse means to move in an opposite direction or contrary to a previous. It consist of two arguments. If the first argument is a list then the second should be an unbound variable or vice versa. The variable must be bound to the value of the list with the elements written in reverse order. The syntax is as follows:

?-reverse(X,Y)

Example 4 shows reverse operation:

Example 4: Illustrate reverse Operation

?- reverse([5,2,3,4],A).

?- reverse(B,[5,2,3,6]).

?- reverse([[dog,cat],[1,2],[bird,mouse],[3,4,5,6]],L).

?- reverse([1,2,3,4],[4,3,2,1]).

?- reverse([1,2,3,4],[3,2,1]).

Figure 10 shows the output of Example 4:

GNU Prolog console

File Edit Terminal Prolog Help

```
GNU Prolog 1.4.5 (64 bits)
Compiled Jul 14 2018, 12:58:46 with cl
By Daniel Diaz
Copyright (C) 1999-2018 Daniel Diaz
| ?- reverse([5,2,3,4],A).

A = [4,3,2,5]

yes
| ?- reverse(B,[5,2,3,6]).

B = [6,3,2,5]

yes
| ?- reverse([[dog,cat],[1,2],[bird,mouse],[3,4,5,6]],L).

L = [[3,4,5,6],[bird,mouse],[1,2],[dog,cat]]

yes
| ?- reverse([1,2,3,4],[4,3,2,1]).

yes
| ?- reverse([1,2,3,4],[3,2,1]).
```

```
no
| ?- |
```

Figure 10: reverse Operation

7.4.8 Append

The concatenating term means two lists are creating a new list by combining. In this the first elements of the first list are followed by the second list. Thus, concatenating [] with [x,y] gives [x,y]. For example, concatenating [x] with [y] gives the list [x y].

The append means to add on, usually to the end of list. It consists of three arguments. In which, the first takes two arguments of lists and the second argument is an unbound variable and the third argument is bound to a list by combining the first two lists. The syntax is as follows:

?-append(X, T, T2).

Example 5 shows append operation:

Example 5: Illustrate append Operation

?- append([1,2,3,4],[5,6,7,8,9],L).

?- append([],[1,2,3],L).

?- append([[a,b,c],d,e,f],[g,h,[i,j,k]],L).

?- append(L1,L2,[1,2,3,4,5]).

?- append(X,[Y|Z],[1,2,3,4,5,6]).

Figure 11 shows the output of Example 5:

GNU Prolog console

File Edit Terminal Prolog Help

```
| ?- append([],[1,2,3],L).

L = [1,2,3]

yes
| ?-  append([[a,b,c],d,e,f],[g,h,[i,j,k]],L).

L = [[a,b,c],d,e,f,g,h,[i,j,k]]

yes
| ?- append([1,2,3,4],[5,6,7,8,9],L).

L = [1,2,3,4,5,6,7,8,9]

yes
| ?- append(L1,L2,[1,2,3,4,5]).

L1 = []
L2 = [1,2,3,4,5] ?

yes
| ?- append(X,[Y|Z],[1,2,3,4,5,6]).

X = []
Y = 1
Z = [2,3,4,5,6] ?
```

7.5 Arithmetic and Search/Execution Order

Prolog has built-in arithmetic operations and an arithmetic evaluator. The arithmetic terms can be written either in the infix notation or in prefix notation: 3 + 4 and + (3, 4). However, Prolog cannot tell when to consider an arithmetic term as a term itself. Therefore,

?- write (3 + 5).

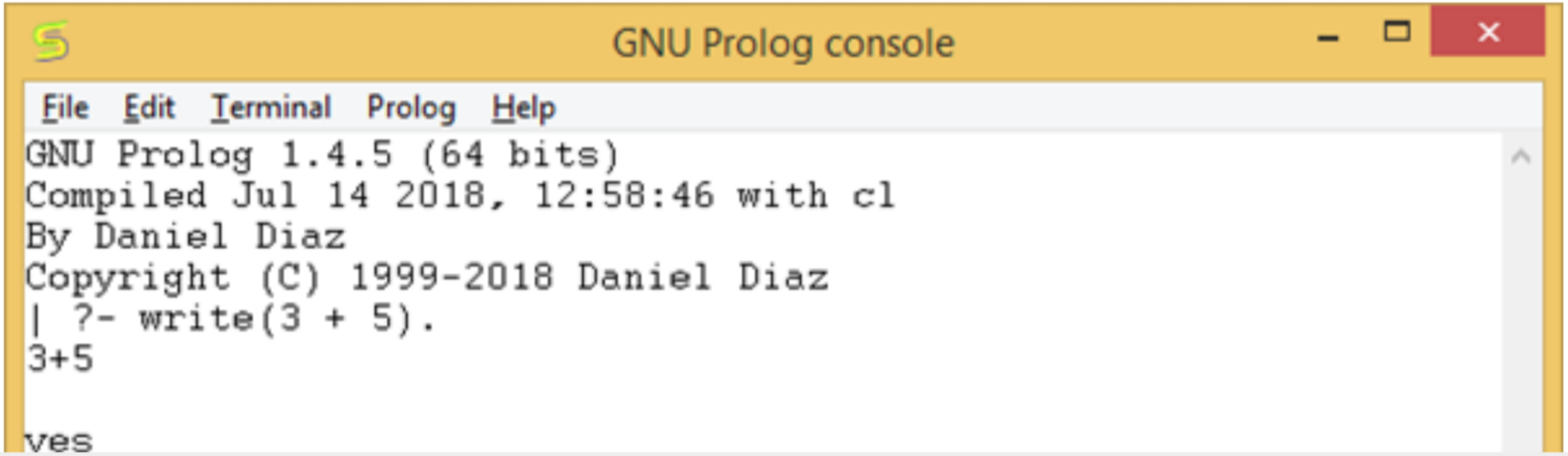
3 + 5

To calculate the arithmetic term, the built-in predicate is required. Therefore, to evaluate 3 + 5, you need to write the following

?- X is 3 + 5, write(X).

X = 8

Figure 12 shows the difference between the two:



```
yes
| ?- X is 3 + 5, write(X).
8

X = 8

yes
| ?- |
```

Figure 12: Arithmetic Execution

Another result of this is that two arithmetic terms may not be equal as terms, even though they have the same value1:

?- 3 + 4 = 4 + 3.

no

To find the equality of values, you need to force the evaluation using is by writing the predicate:

valequal(Term1, Term2) :-

X is Term1, Y is Term2, X = Y.

This can be used as:

?- valequal(3 + 4, 4 + 3).

yes

In prolog 'is' has a special functionality in evaluating arithmetic expressions. But with condition that the expression should be on the right side of 'is' otherwise it will give an error.

Let s look at a few more arithmetic examples:

?- X is 3+2. // expression on right side of 'is'

X = 5.

?- 3+2 is X. // expression on left side of 'is'

ERROR: is/2: Arguments are not sufficiently instantiated

?- X = 3+2. // just instantiate variable X to value 3+2

X = 3+2.

?- 3+2 = X.

X = 3+2.

?- X is +(3,2).

X = 5.

?- 5 is 3+2.

true.

?- 3+2 is 5.

false.

?- X is 3*2.

```
?- X is 3*2.

X = 6.

?- X is 3-2.

X = 1.

?- X is -(2,3).

X = -1.

?- X is 5-3-1.

X = 1.

?- X is -(5,3,1).

ERROR: is/2: Arithmetic: `(-)/3' is not a function

?- X is -(-(5,3),1).

X = 1.

?- X is 5-3-1.

X = 1

?- X is 3/5.

X = 0.6.

?- X is 3 mod 5.

X = 3
```

```
X = 3.

?- X is 5 mod 3.

X = 2.

?- X is 5^3.

X = 125.

?- X is (5^3)^2.

X = 15625.

?- X = (5^3)^2.

X = (5^3)^2.

?- 25 is 5^2.

true.

?- Y is 3+2*4-1.

Y = 10.

?- Y is (3+2)*(4)-(1).

Y = 19.

?- Y is -(*(+(3,2),4),1).

Y = 19.
```

?- X is 3*2, Y is X*2.

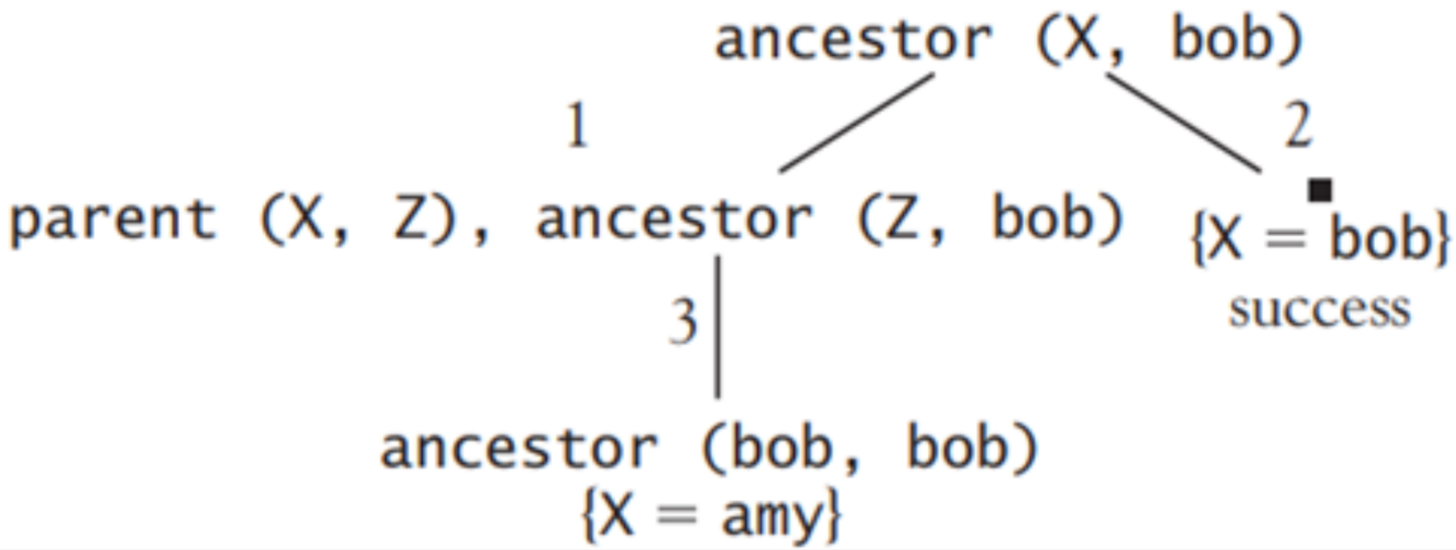
X = 6,

Y = 12.

In Prolog, resolution is implemented in a linear manner where the goals are replaced left to right and considering clauses in the database in top-to-bottom order. Therefore, the search strategy results in a depth-first search on a tree of possible choices. For example, let us discuss the following clasues:

1. ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
2. ancestor(X, X).
3. parent(amy, bob).

With the goal ancestor (X,bob), Prolog's search strategy is left to right and depth first on the tree of subgoals shown in Figure 13. In this figure the edges are labelled by the number of the clause above used by Prolog for resolution, and instantiations of variables are written in curly brackets.



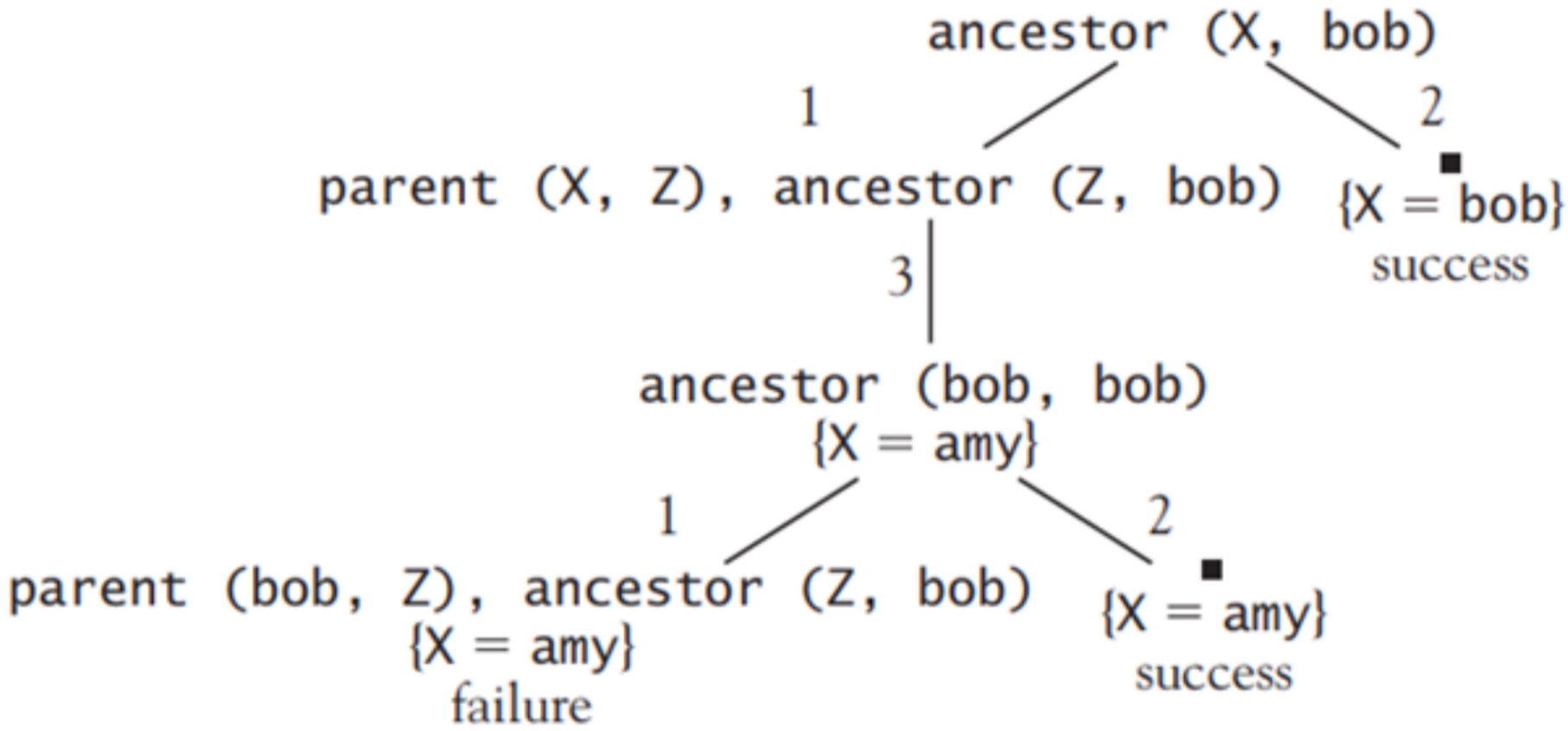


Figure 13: Prolog Search Tree

Leaf nodes in this tree occur when there is no match for the leftmost clause or when all clauses have been eliminated. Whenever failure happens, the user indicates a continued search with a semicolon, Prolog backtracks up the tree to find further paths to a leaf and releases instantiations of variables. Therefore, in tree after the solution X = amy is found, this instantiation of X will be released, and the new path to a leaf with X = bob will be found.

The depth-first strategy is efficient because it can be implemented in a stack-based or recursive fashion. However, in this case it also possible that solutions may not be found if the search tree has branches of infinite depth.

7.6 Imperative Control Flow

Most conventional programming languages provide loops that allows a set of instructions to be executed repeatedly either a fixed number of times or until a given condition is met. Prolog has no looping facilities; similar effects can be obtained that enable a sequence of goals to be evaluated repeatedly. It can be done using backtracking, recursion, built-in predicates, or a combination of these.

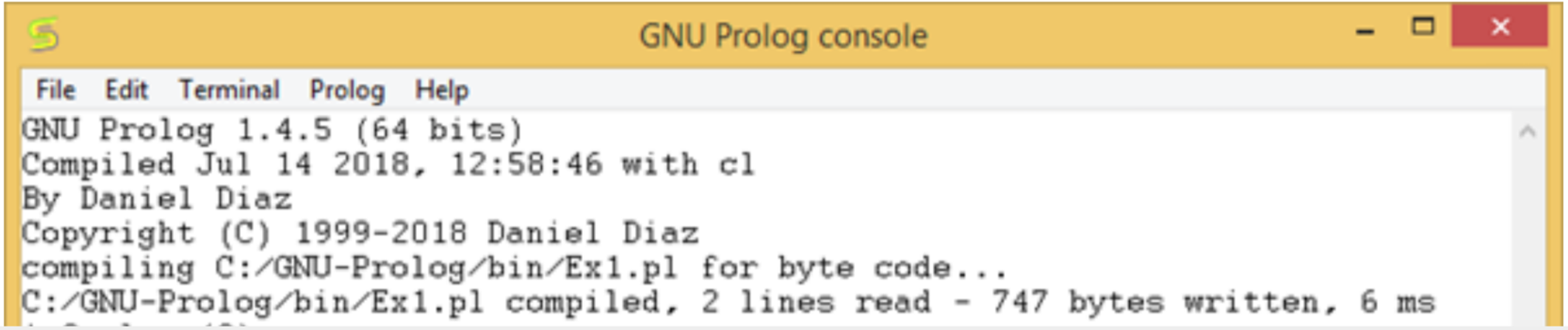
7.6.1 Loop for a Specific Number of Time

Most programming languages use 'for loops' which enable a set of instructions to be executed a fixed number of times. It is not available in Prolog but it can be obtained using recursion, as shown in Example 6:

Example 6: Looping For a Specific Time

```
loop(0).

loop(N):-N>0,write('The value is: '),write(N),nl,

 M is N-1,loop(M).
```

Figure 14 shows the output of Example 6:



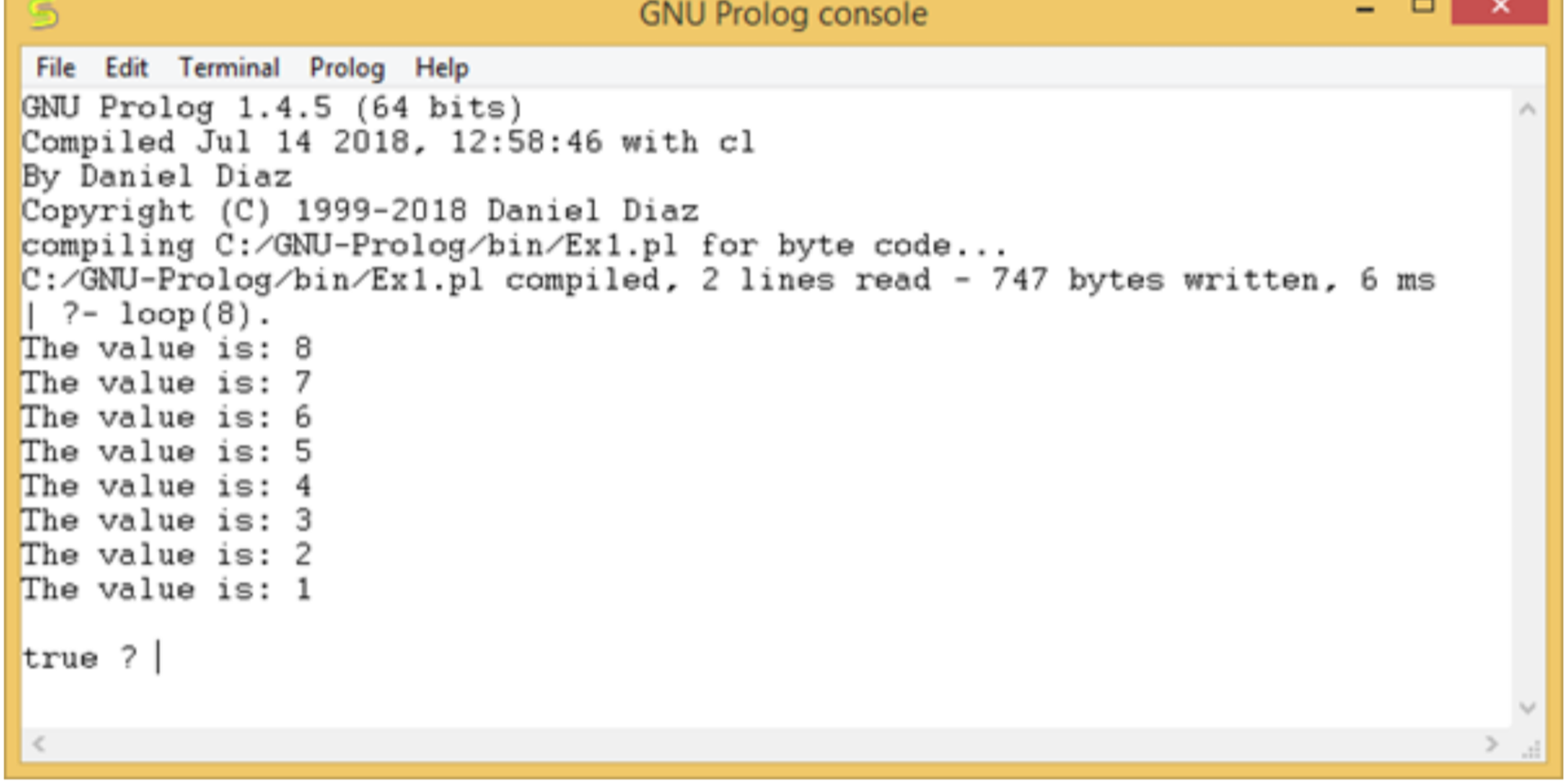


Figure 14: Output for Looping

Example 7 is used to provide integers from First to Last inclusive:

Example 7: Integers First To Last

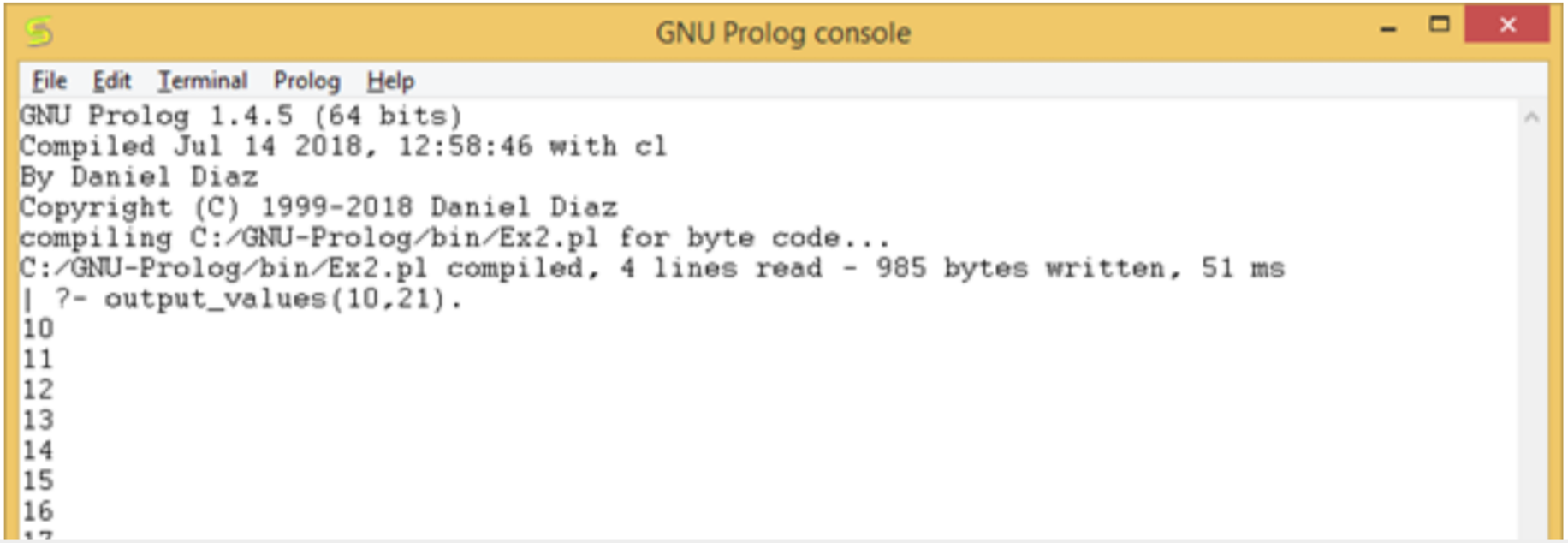
/* output integers from First to Last inclusive */

Example 7 is used to provide integers from First to Last inclusive:

Example 7: Integers First To Last

```
/* output integers from First to Last inclusive */

output_values(Last,Last):- write(Last),nl,

 write('end of example'),nl.

output_values(First,Last):-First=\=Last,write(First),

 nl,N is First+1,output_values(N,Last).
```

Figure 15 shows the output of Example 7:



```
GNU Prolog 1.4.5 (64 bits)
Compiled Jul 14 2018, 12:58:46 with cl
By Daniel Diaz
Copyright (C) 1999-2018 Daniel Diaz
compiling C:/GNU-Prolog/bin/Ex2.pl for byte code...
C:/GNU-Prolog/bin/Ex2.pl compiled, 4 lines read - 985 bytes written, 51 ms
| ?- output_values(10,21).
10
11
12
13
14
15
16
17
18
19
20
21
end of example

true ?

(15 ms) yes
| ?-
```

Figure 15: Integers First To Last

7.6.2 Loop till a Condition Is Satisfied

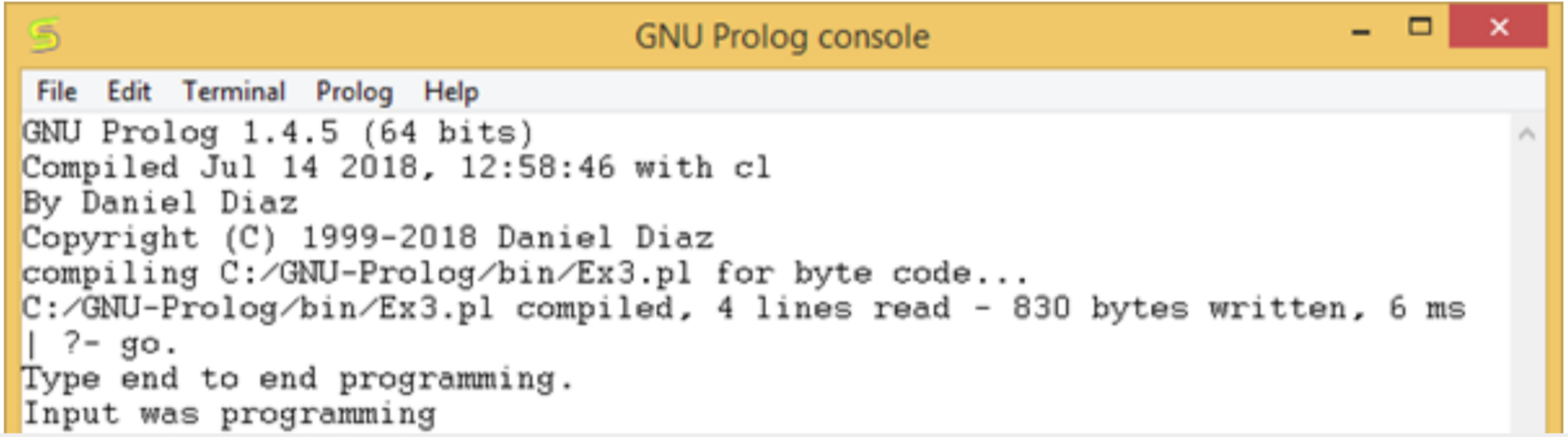
7.6.2 Loop till a Condition Is Satisfied

Most programming languages use 'until loops' which enable a set of instructions to be executed a fixed number of times. It is not available in Prolog but it can be obtained using recursion, as shown in Example 8:

Example 8: Looping Until Condition is Fulfilled

```
go:-loop(start). /* start is a dummy value used to get

 the looping process started.*/

loop(end).

loop(X):-X\=end,write('Type end to end'),read(Word),

 write('Input was '),write(Word),nl,loop(Word).
```

Figure 16 shows the output of Example 8:



GNU Prolog console

File Edit Terminal Prolog Help

```
GNU Prolog 1.4.5 (64 bits)
Compiled Jul 14 2018, 12:58:46 with cl
By Daniel Diaz
Copyright (C) 1999-2018 Daniel Diaz
compiling C:/GNU-Prolog/bin/Ex3.pl for byte code...
C:/GNU-Prolog/bin/Ex3.pl compiled, 4 lines read - 830 bytes written, 6 ms
| ?- go.
Type end to end programming.
Input was programming
Type end to end logic.
Input was logic
Type end to end descriptive.
Input was descriptive
Type end to end end.
Input was end

true ?

(31 ms) yes
| ?- |
```

Figure 16: Output of Example 8

7.7 Database Manipulation

A database is a collection of information which allow organisation to access, manage and update the information. Typically, system databases consist of aggregations of data records about transactions with specific customers. Several databases consist of multiple tables which is capable to include different fields. For example, an organisation required database for products, employees, and financial records. Each of these database have different fields which might be relevant to the information stored in the database.

A data manipulation provide sublanguage for database such as SQL. On the other side, Prolog is effective language for database queries. A prolog programming language allow programmer to modify the content of the database during run-time. It also programmer to use atoms, integers and reals while the writing the program. Once it gets a database language then it is equivalent to a subset of SQL. Prolog has three database manipulation commands namely retract, asserta, and assertz.

- **Asserta:** It means adds a new fact or clause to the database. The asserted term is added as the last fact with the same key predicate. For example, 'a' is the first letter of the alphabet and 'z' being the last then it should remind where to add new fact in the database. It is similar as assert but adds a fact at the beginning of the database.
- **Assertz:** It insert assert fact at the end of the database.
- **Retract :** It removes fact from the database.

Table 3 shows the use of the assertz, asserta, and retract predicates for changing the database:

Table 3: Database Manipulation

Code	Output
?- listing(mypred). yes	Initially there is no predicate mypred in the database
?- assertz(mypred(first)). yes	Four mypred clauses are added to the database.

?- assertz(mypred(first)). yes ?- assertz(mypred(second)). yes ?- assertz(mypred(third)). yes ?- assertz(mypred(fourth)). yes	Four mypred clauses are added to the database.
?- listing(mypred). /* mypred/1 */ mypred(first). mypred(second). mypred(third). mypred(fourth). yes	The four mypred clauses are now in the database
?- asserta(mypred(new1)). yes	A new mypred clause is added to the database, using the asserta predicate

?- asserta(mypred(new1)). yes	A new mypred clause is added to the database, using the asserta predicate
?- listing(mypred). /* mypred/1 */ mypred(new1). mypred(first). mypred(second). mypred(third). mypred(fourth). yes	This shows that asserta places the new clause above the other mypred clauses in the database
?- assertz(mypred(new2)). yes	A further mypred clause is added to the database using the assertz predicate
?- listing(mypred). /* mypred/1 */ mypred(new1). mypred(first).	This shows that assertz places the new clause below the other mypred clauses in the database

mypred(first). mypred(second). mypred(third). mypred(fourth). mypred(new2). yes	
?- mypred(X). X = new1 ; X = first ; X = second ; X = third ; X = fourth ; X = new2	This shows that retrieving mypred clauses from the database will give the first one listed by the listing predicate Subsequent backtracking will obtain the remaining clauses in top to bottom order
?- retract(mypred(first)). yes	One of the mypred clauses is removed from the database using the retract predicate
?- listing(mypred).	This shows that it has successfully been removed

X = third ; X = fourth ; X = new2	
?- retract(mypred(first)). yes	One of the mypred clauses is removed from the database using the retract predicate
?- listing(mypred). /* mypred/1 */ mypred(new1). mypred(second). mypred(third). mypred(fourth). mypred(new2). yes	This shows that it has successfully been removed

7.8 Prolog Deficiencies

The goal of logic programming was to allow the programmer to specify only the properties of a solution and to let the language implementation provide the actual method for computing the solution from its properties. In declarative programming the perspective is a program describes what a solution to a given problem is and not how that problem is solved.

Logic programming language have fulfilled this goal. However, the nature of the algorithms used by logic programming systems, such as resolution, unification, and depth-first search, have introduced many pitfalls into the specifics of writing programs.

Some of those pitfalls are as follows:

- **Resolution order control:** Prolog always matches in the same order where the user can control the ordering. It allows explicit control of backtracking using the cut operator -- "!". It is detrimental to one of the important advantages of logic programming. The programs do not specify how solutions are to be found.
- **The closed-world assumption**: In Prolog, the truths are those that can be proved using its database. If there is insufficient information in the database to prove a query, it is not actually false, it fails.
- **The negation problem**: The negation problem states that two atoms are not equal is not straightforward in Prolog. A bad solution for this problem can be adding facts stating that they are not the same. Another solution can be the state in the goal that X must not be the same as Y

sibling(X, Y) :- parent(M, X), parent(M, Y), not(X=Y)

However, not operator is not logical NOT, not(not(some_goal)) is not equivalent to some_goal.

- **Intrinsic limitations**: Prolog does not provide any method to sort. However, it simply allows to enumerate all permutations of the given list until it happens to create the one that has the list in sorted order. You need to specify the details of how that sorting can be done.

7.9 Prolog Examples

In this section, you will see some of the basic Prolog examples:

Example 9: In this example, the first three lines are facts and the last line is a rule that allows you to add in logic such as querying facts and other rules.

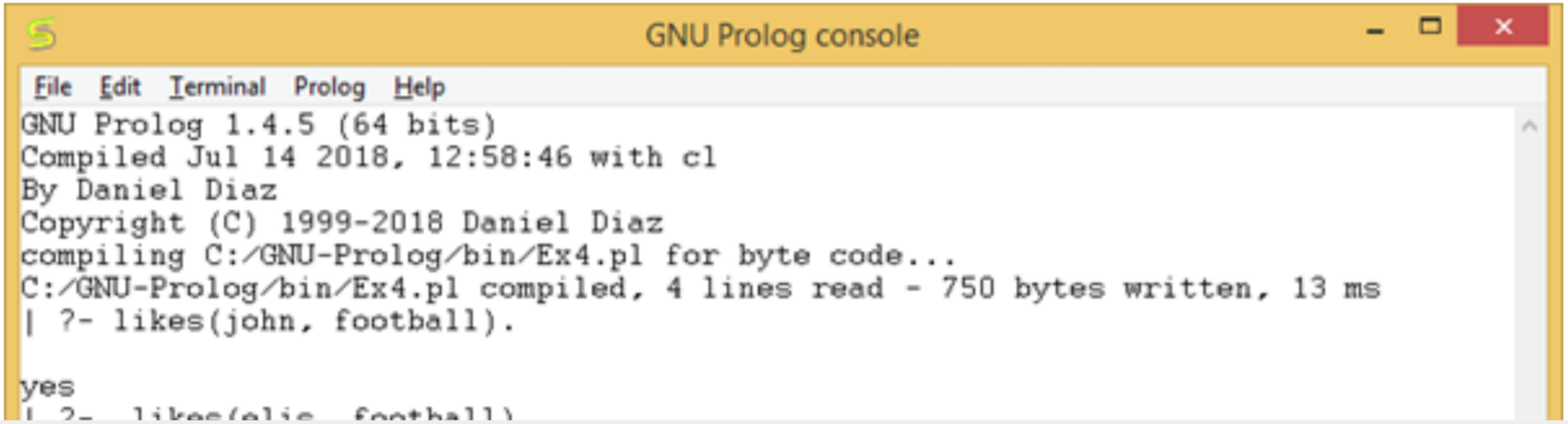
likes(john, football).

likes(harry, football).

likes(elis, badminton).

friend(X, Y) :- likes(X, Z), likes(Y, Z).

Figure 17 shows the output of example 9:



```
yes
| ?-  likes(elis, football).

no
| ?- friend(john,elis).

no
| ?- friend(john,harry).

yes
| ?- |
```

Figure 17: Output of Example 9

Example 10 : Food

Facts	English meanings of Facts, Rules & Goals
food(vadapav).	// vadapav is a food
food(sandwich).	// sandwich is a food
food(biryani).	// biryani is a food
lunch(sandwich).	// sandwich is a lunch
dinner(biryani).	// biryani is a dinner

dinner(biryani).	// biryani is a dinner
Rules	
meal(X) :- food(X).	// Every food is a meal OR Anything is a meal if it is a food
Queries / Goals & answers	
?- food(biryani).	// Is biryani a food?
true.	Answer : true Explanation : Here prolog will return 'true or yes'. Because first, prolog interpreter will trace through the facts and rules in top-down manner and when it can find the match it will provide the answer and in this case it can find the exact match.

?- meal(X), lunch(X).	// Which food is meal and lunch? OR What is both meal and lunch?
X = sandwich.	Answer : X = sandwich. Explanation : Here in this query we have provided two subgoals where "," comma means 'and'. Prolog always tries to satisfy subgoals in left-to-right manner, so first try to get left most goal i.e. meal(X). But meal(X) rule says - X is a meal if X is a food. So, now we will look for food(X). Here X is a variable and it can bound with any related value. So, in food(X) - X can be vadapav, sandwich or biryani as per our facts. But second goal says that X should also be lunch. Now we look for X value in lunch(X) and i.e. sandwich. So now we find which food(X) values matches with lunch(X) and the answer is sandwich.

?- dinner(sandwich).	// Is sandwich a dinner?
false.	Answer : false. Explanation : In this case prolog will find the 'dinner' predicate and will match the argument inside the bracket. But it will return 'false or no' since it cannot find the match.

Example 11:

Facts	**English meanings of Facts, Rules & Goals**
studies(chaitanya, maths).	// chaitanya studies maths
studies(siddhi, maths).	// siddhi studies maths
studies(jayesh, physics).	// jayesh studies physics
studies(arvind, chemistry).	// arvind studies chemistry

studies(arvind, chemistry).	// arvind studies chemistry
teaches(kalyani, maths).	// kalyani teaches maths
teaches(chirag, physics).	// chirag teaches physics
teaches(janhavi, chemistry).	// janhavi teaches chemistry
Rules	
professor(X, Y) :- teaches(X, C),	// X is a professor of Y if X teaches C and Y studies C.
studies(Y, C).	(here X is a professor, Y is a student and C is a course and X, Y, C are variables)
Queries / Goals & answers	
?- studies(chaitanya, What).	// chaitanya studies what? OR What does Chaitanya study?
What = maths.	Answer : maths. Explanation : Here in query 'What' is a variable since it starts from uppercase

What = maths.	Answer : maths. Explanation : Here in query 'What' is a variable since it starts from uppercase letter. Again prolog will match the values from given prolog database in top-down manner and it will check the query from left-to-right. First, it will find the predicate studies, then chaitanya and if any value for 'What' variable would be matched then variable binding will occur. As per that prolog will return 'What = maths' as an answer to the query.
?- professor(kalyani, Students).	// Who are the students of professor kalyani. OR kalyani is a professor of which students.
Students = chaitanya;	Answer : Students = chaitanya; Students = siddhi.
Students = siddhi.	(here in query Students is a variable) Explanation : Here we have defined professor rule in the prolog program that - 'professor(X, Y) :- teaches(X, C), studies(Y, C). Where X is a professor, Y is a student and C is a course'. So, as per that rule and given query, prolog has the value of X i.e. kalyani and now prolog will try to find values for Students. For that prolog reads the right hand side of the rule and from rule and value of X it can interpret - teaches(kalyani, C),

	side of the rule and from rule and value of X it can interpret - teaches(kalyani, C), studies(Y,C). Now, as prolog tries to satisfy things from left-to-right, it finds the match for 'teaches' fact and from that it has value of C i.e. maths. Now it has - teaches(kalyani, maths), studies(Y, maths). Now it tries to find fact for studies and from given facts prolog can find two values for Y i.e. Chaitanya and siddhi. Now, prolog may return only 'Students = chaitanya'. as an answer but we know that there more than one student as this query's answer. So, to get the next solution for the query, we need to type ";". But after prolog finds all the solutions and then even you will type ';' to get next solution, prolog will return 'false' because it can not find further solutions.

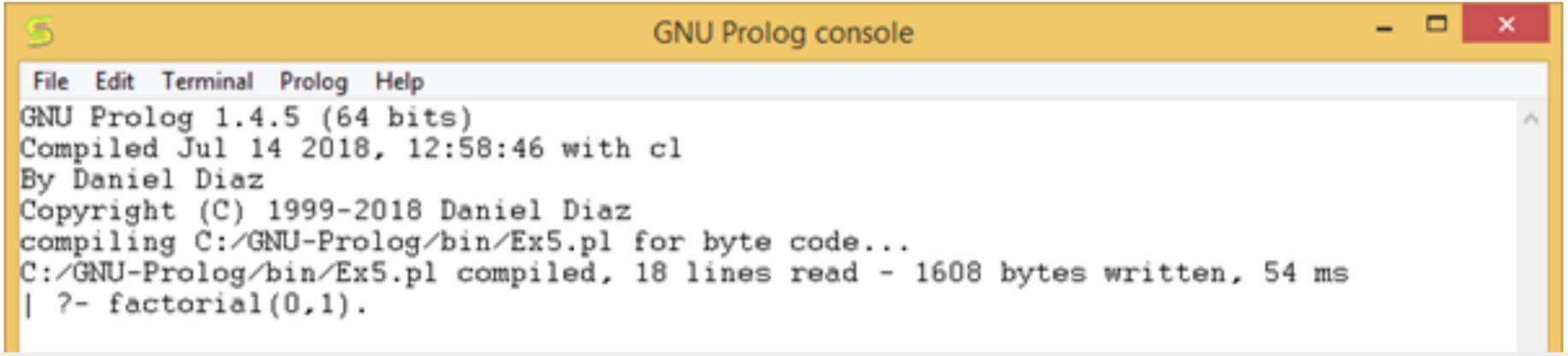
Example 10: This example consists of two clauses where the first clause is a unit clause, having no body. The second is a rule because it has a body. The body of second clause is on the right side of the ':-' which can be read as "if". The body consists of literals separated by commas ',' each of which can be read as "and".

A declarative reading of the first (unit) clause says that "the factorial of 0 is 1" and the second clause declares that "the factorial of N is F if N>0 and N1 is N-1 and the factorial of N1 is F1 and F is N*F1"..

```
/* Factorial Definitions */

factorial(0,1).

factorial(N,F) :-

  N>0,

  N1 is N-1,
```

```
  N1 is N-1,

  factorial(N1,F1),

  F is N * F1.

/*-----------------------------*/

factorial(0,F,F).

factorial(N,A,F) :-

   N > 0,

   A1 is N*A,

   N1 is N -1,

   factorial(N1,A1,F).
```

Figure 18 shows the output of example 10:



```
| ?- factorial(0,1).

true ?

yes
| ?- factorial(5,120).

true ?

yes
| ?- factorial(5,12).

no
| ?- factorial(8,1,F).

F = 40320 ?

yes
| ?- |
```

Figure 18: Output of Example 10

Example 11: The task of this puzzle is to move N disks from the left peg to the right peg using the centre peg as an auxiliary holding peg. A larger disk cannot be placed upon a smaller disk. Figure 19 depicts the starting setup for N=3 disks:

Example 11: The task of this puzzle is to move N disks from the left peg to the right peg using the centre peg as an auxiliary holding peg. A larger disk cannot be placed upon a smaller disk. Figure 19 depicts the starting setup for N=3 disks:

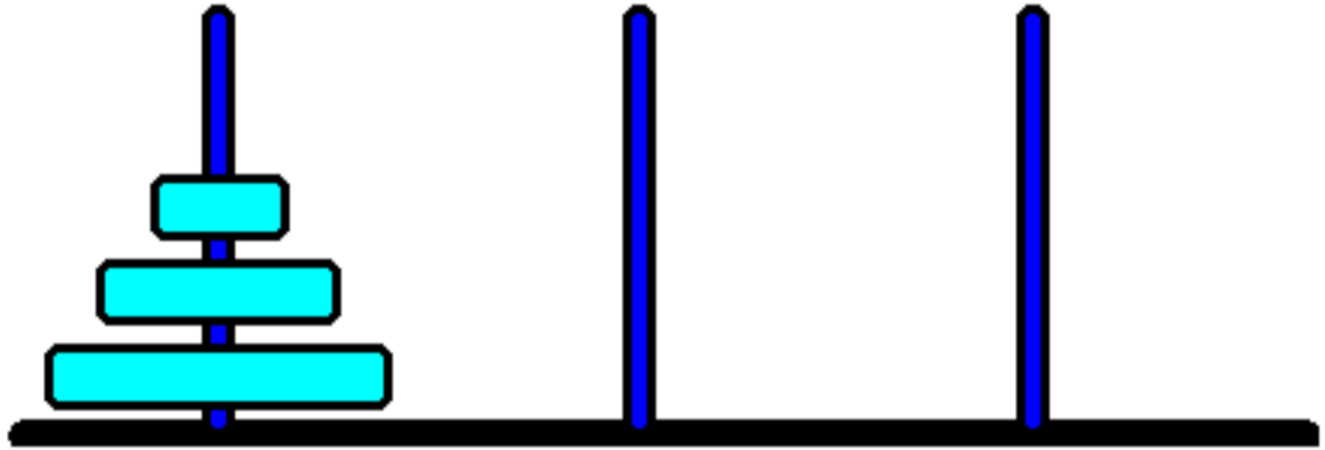
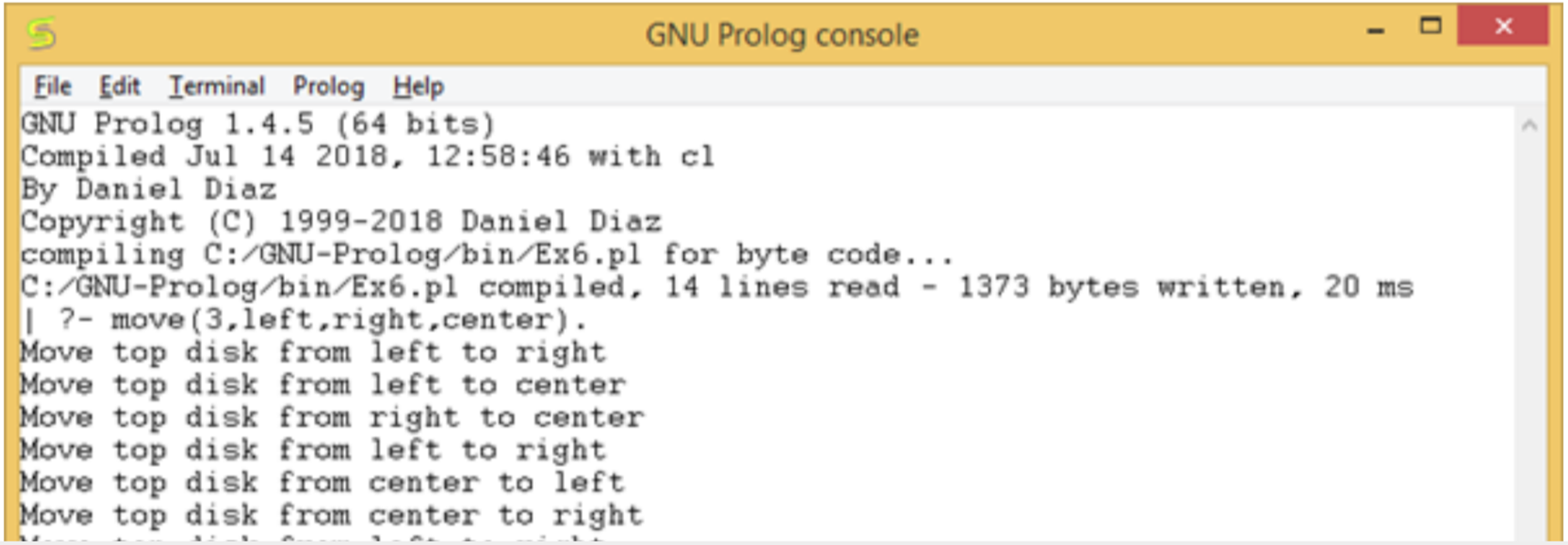


Figure 19: Tower of Hanoi

```
/* Towers of Hanoi puzzle */

move(1,X,Y,_) :-

   write('Move top disk from '),

   write(X),

   write(' to '),

   write(Y),
```

```
    nl.

move(N,X,Y,Z) :-

    N>1,

    M is N-1,

    move(M,X,Z,Y),

    move(1,X,Y,_),

    move(M,Z,Y,X).
```

Figure 20 shows the output of example 11:



```
Move top disk from left to center
Move top disk from right to center
Move top disk from left to right
Move top disk from center to left
Move top disk from center to right
Move top disk from left to right

true ?

yes
| ?- |
```

Figure 20: Output of Example 11

Let us quickly recapitulate the following discussion points:

Example 12: Knowledge Base 1

Knowledge Base 1 (KB1) is simply a collection of facts.

Facts are used to state things that are unconditionally true of some situation of interest.

E.g. Assume that Ria, Jyoti, and Kalpana are women, that Jyoti plays sitar, and that a party is taking place, using the following five facts:

woman(Ria).

woman(Jyoti).

woman(Kalpana).

playsSitar(Jyoti).

```
playsSitar(Jyoti).
party.
```

```
?- woman(Ria).
true.
?- woman(Jyoti).
true.
?- woman(Kalpana).
true.
?- playsSitar(Jyoti).
true.
?- party.
true.
```

Example 13: Knowledge Base 2

```
happy(Kalpana).
listens2Music(Ria).
```

listens2Music(Kalpana):- happy(Kalpana).

playsSitar(Ria):- listens2Music(Ria).

playsSitar(Kalpana):- listens2Music(Kalpana).

The first rule says that Kalpana listens to music if she is happy, and the last rule says that Kalpana plays sitar if she listens to music.

More generally, the :- should be read as “if”, or “is implied by”.

The part on the left hand side of the :- is called the head of the rule, the part on the right hand side is called the body.

Head :- Body

A general rules say that if the body of the rule is true, then the head of the rule is true too.

If a knowledge base contains a rule i.e. head :- body, then, Prolog knows that body follows from the information in the knowledge base, and it can infer head. This fundamental deduction step is called modus ponens.

Example 14: Knowledge Base 3

happy(Vivek).

```
listens2Music(Bhushan).

playsSitar(Vivek):-listens2Music(Vivek),happy(Vivek).

playsSitar(Bhushan):- happy(Bhushan).

playsSitar(Bhushan):- listens2Music(Bhushan).

playsSitar(Bhushan):- happy(Bhushan); listens2Music(Bhushan).
```

The most important thing to note is the comma, that separates the goal listens2Music(Vivek) and the goal happy(Vivek) in the rule's body.

So this rule says: "Vivek plays sitar if he listens to music and he is happy".

There is another way of expressing disjunction in Prolog. We could replace the pair of rules given above by the single rule

```
playsSitar(Bhushan):-happy(Bhushan); listens2Music(Bhushan).
```

Example 15: Knowledge Base 4 : (Variable instantiation)

```
woman(Ria).

woman(Jyoti).

woman(Kalpana).
```

```
loves(pumpkin,honey_bunny).

loves(honey_bunny,pumpkin).
```

```
?- woman(X).
```

The X is a variable.

Now a variable isn't a name, rather it's a placeholder for information.

That is, this query asks Prolog: tell me which of the individuals you know about is a woman.

Prolog answers this query by working its way through KB4, from top to bottom, trying to unify (or match) the expression woman(X) with the information KB4 contains.

Now the first item in the knowledge base is woman(Ria) . So, Prolog unifies X with Ria , thus making the query agree perfectly with this first item. (Incidentally, there's a lot of different terminology for this process: we can also say that Prolog instantiates X to Ria, or that it binds X to Ria .) Prolog then reports back to us as follows:

```
X = Ria
```

That is, it not only says that there is information about at least one woman in KB4, it actually tells us who she is. It didn't just say yes, it actually gave us the variable binding (or variable instantiation) that led to success.

But that's not the end of the story. The whole point of variables is that they can stand for, or unify with, different things. And there is information about other women in the knowledge base. We can access this information by typing a semicolon:

```
loves(Vivek,Ria).

loves(Makarand,Ria).

loves(pumpkin,honey_bunny).

loves(honey_bunny,pumpkin).
```

```
?-  woman(X).
```

The X is a variable.

Now a variable isn't a name, rather it's a placeholder for information.

That is, this query asks Prolog: tell me which of the individuals you know about is a woman.

Prolog answers this query by working its way through KB4, from top to bottom, trying to unify (or match) the expression woman(X) with the information KB4 contains.

Now the first item in the knowledge base is woman(Ria) . So, Prolog unifies X with Ria , thus making the query agree perfectly with this first item. (Incidentally, there's a lot of different terminology for this process: we can also say that Prolog instantiates X to Ria, or that it binds X to Ria .) Prolog then reports back to us as follows:

```
X  =  Ria
```

That is, it not only says that there is information about at least one woman in KB4, it actually tells us who she is. It didn't just say yes, it actually gave us the variable binding (or variable instantiation) that led to success.

But that's not the end of the story. The whole point of variables is that they can stand for, or unify with, different things. And there is information about other women in the knowledge base. We can access this information by typing a semicolon:

```
X  =  Ria  ;
```

Remember that **;** means **or**. Therefore this query means: are there any alternatives ? So Prolog begins working through the knowledge base again (it remembers where it got up to last time and starts from there) and sees that if it unifies X with Jyoti , then the query agrees perfectly with the second entry in the knowledge base. So it responds:

Remember that **;** means **or**. Therefore this query means: are there any alternatives ? So Prolog begins working through the knowledge base again (it remembers where it got up to last time and starts from there) and sees that if it unifies X with Jyoti , then the query agrees perfectly with the second entry in the knowledge base. So it responds:

```
X = Ria ;

X = Jyoti
```

Example 16: Knowledge Base 5

```
loves(Vivek,Ria).

loves(Makarand,Ria).

loves(pumpkin,honey_bunny).

loves(honey_bunny,pumpkin).


jealous(X,Y):-  loves(X,Z),  loves(Y,Z).
```

KB5 contains four facts about the loves relation and one rule.

It contains three variables (note that X , Y , and Z are all upper-case letters). What does it say?

Suppose we pose the query:

```
?-  jealous(Makarand,W).
```

This query asks: can you find an individual W such that Makarand is jealous of W ?

```
loves(Makarand,Ria).

loves(pumpkin,honey_bunny).

loves(honey_bunny,pumpkin).


jealous(X,Y):-  loves(X,Z),  loves(Y,Z).
```

KB5 contains four facts about the loves relation and one rule.

It contains three variables (note that X , Y , and Z are all upper-case letters). What does it say?

Suppose we pose the query:

```
?-  jealous(Makarand,W).
```

This query asks: can you find an individual W such that Makarand is jealous of W ?

Vivek is such an individual.

If you check the definition of jealousy, you'll see that Makarand must be jealous of Vivek, because they both love the same woman, namely Ria. So Prolog will return the value

```
W  =  Vivek
```

7.10 Recap and Questions

7.10.1 Recap

Let us quickly recapitulate the following discussion points:

- Logic Programming is different from traditional programming languages such as C, C++ and Java.
- Prolog was one of the first logic programming languages.
- Prolog programs are written in a special format that is known as clausal form.
- Unification in Prolog is possible if two expressions can be made to have “the same structure” through variable instantiation.
- In Prolog, a list refers to an ordered sequence of elements. Lists are a common data structure in computer programming.
- Prolog has built-in arithmetic operations and an arithmetic evaluator.
- Prolog has three database manipulation commands namely retract, asserta, and assertz.

7.10.2 Test Your Skills- Multiple Choice Questions

1. Which of the following operation accepts two arguments where the first argument can be any term and the second argument must be a list?

1. Which of the following operation accepts two arguments where the first argument can be any term and the second argument must be a list?

a. member
b. length
c. reverse
d. append

2. Which of the following operation defines the size of an object or distance from one point to another?

a. member
b. length
c. reverse
d. append

3. Which of the following operation means to move in an opposite direction or contrary to a previous?

a. member
b. length
c. reverse
d. append

4. Which of the following is used to place the new clause above the other mypred clauses in the database?

a. member
b. length
c. asserta
d. append

5. Which of the following is used place the new clause below the other mypred clauses in the database?

a. assertz
b. length
c. asserta
d. append

3. Which of the following is used place the new clause below the other myprel clauses in the database?

a. assertz
b. length
c. asserta
d. append

7.10.3 Fill in the Blanks

1. A _____ refers to a symbol which represents a value like "a" or "b,".
2. _____ are constants which do not have a numerical value.
3. Prolog programs are written in a special format that is known as _____ form.
4. _____ in Prolog is possible if two expressions can be made to have “the same structure” through variable instantiation.
5. In Prolog, a _____ refers to an ordered sequence of elements.

7.10.4 Theoretical Questions

1. Explain the four database manipulation commands.
2. Explain the Prolog deficiencies.
3. Describe the control flow in Prolog.
4. Describe the list operations.
5. Explain the unification and resolution.

Chapter 8: Concurrency

Learning Objectives

At the end of this module, you will be able to:

- Describe concurrent programming
- Explain message passing
- Explain multi-threaded programs
- Describe how to implement synchronization
- Discuss interthread communication

8.1 Introduction

A program is said to be concurrent if it has more than one active execution context. The concurrent word is used to characterise any system that has two or more tasks may be underway at the same time. A concurrent system can be parallel if more than one task can be physically active at once. It needs more than one processor. In a concurrent program, the term thread defines the present entity which the programmer might think as running concurrently with other threads. In many systems, threads given to a program are developed on top of one or more processes which is provided by the operating system.

In this chapter, you will learn about concurrent programming and message passing. You will also learn about multi-threaded programs. You will learn about how to implement synchronization. At the end of this chapter you will learn about interthread communication.

8.2 Concurrent Programming Fundamentals

In programming concurrent refers to computing model in which more than one processors at a time can execute instructions simultaneously for better performance. In another word, concurrent means which occurs when something else happens. Thus, the tasks are broken into sub-types to perform simultaneously in which each task are assigned to different processors. It helps in several techniques, for example, coordinating the execution of processes, memory allocation and execution scheduling for maximizing throughput. The principles of concurrency are when both interleaved and overlapped processes are viewed as examples of concurrent processes because they both present at the same time. Thus, their processing speed cannot be calculated at the time. It depends on the activities of other processes, methods of operating system to handle interrupts and scheduling policies of the operating system.

Sometimes, concurrent processing works similar to parallel processing. Parallel processing provides genuine simultaneous execution. It also arises the issue different processors which might run at different speeds, but again this problem is mirrored in concurrency because different processes are running at a different speed. Interrupt-driven I/O developed concurrency within the operating system because an interrupt can occur at an arbitrary time. It includes a control which is already present in the OS. Thus, the interrupt handles main bulk of the OS function as concurrent threads of control.

In a concurrent program, the term thread defines the present entity which the programmer might think as running concurrently with other threads. In many systems, threads given to a program are developed on top of one or more processes which is provided by the operating system. OS model able to differentiate between a heavyweight process and lightweight processes which might share an own address space. In 1980s lightweight processes were added to most variants of Unix to accommodate with shared-memory multiprocessors. Many OS call lightweight processes as threads. The fundamental problem in concurrency is processes. Where each process interferes with each other while accessing a shared global resource like

```
X = getchar();

Y = in;

putchar(Y);
```

putchar(Y);

Suppose two processes P1 and P2 both are executing code at the same time with the following conditions:

- P1 enters into code, after reading the character X into in is interrupted.
- P2 enters into code, its runs completed after reading and displaying the character Y.
- P1 is resumed, but X now contains the character y thus P1 displays the wrong character.

The problem occurred due to shared global variable i.e. X. P1 sets X, but still show subsequently lost during the execution of P2. However, the general solution allows only one process at a time to enter the code and such code are known as a critical section. In critical section when one process is inside then other processes should be prevented from entering into that section and this phenomenon is known as mutual exclusion.

The advantages and disadvantages of the concurrency have shown in Table 1:

Table 1: Advantages and Disadvantages of Concurrency

Advantages	**Disadvantages**
Run multiple applications at the same time.	Protect multiple applications from one another.
Unused resources can be used for other applications by one application.	Coordinate with multiple applications via additional mechanisms.
Without concurrency, every application runs completely before the next one can start running.	In operating systems performance required overheads and complexities to switch among other applications.

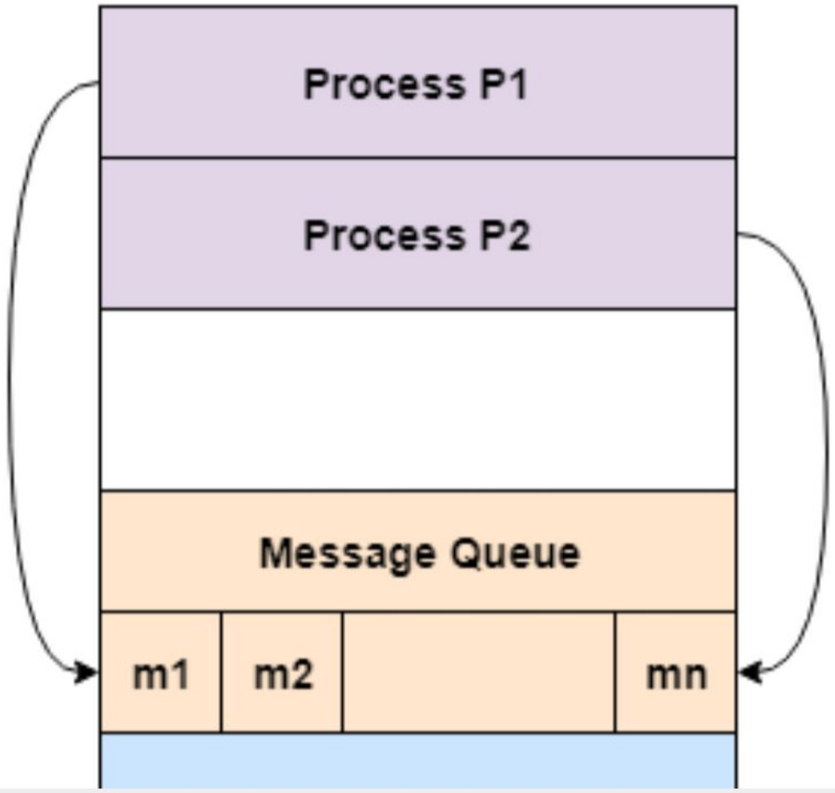
Unused resources can be used for other applications by one application.	Coordinate with multiple applications via additional mechanisms.
Without concurrency, every application runs completely before the next one can start running.	In operating systems performance required overheads and complexities to switch among other applications.
Enables better performance by the operating system.	Several applications running concurrently can degrade performance.

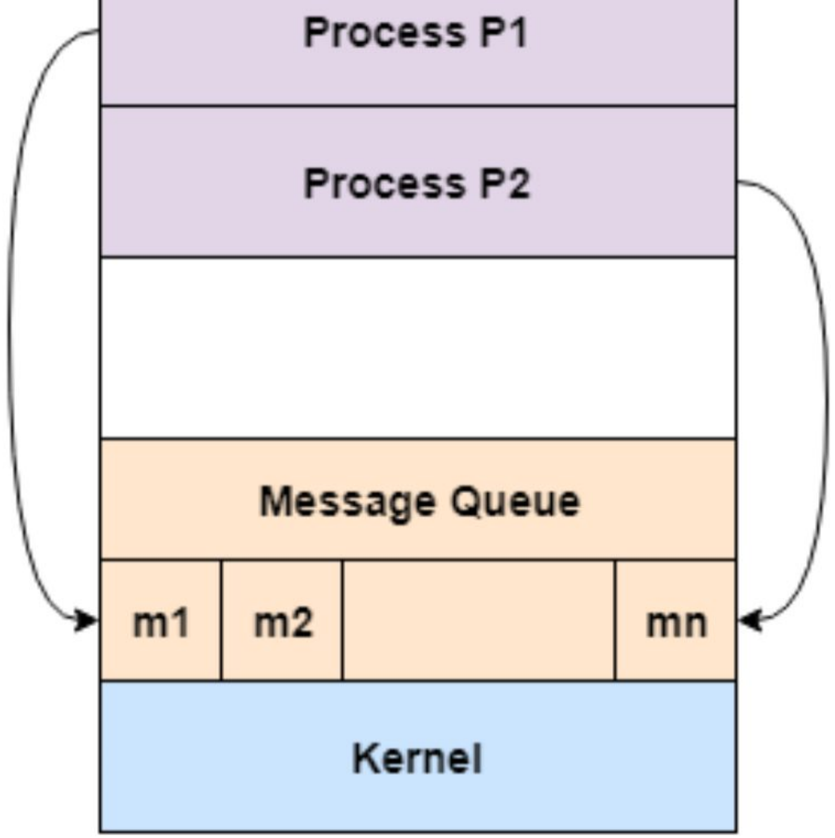
Several issues of concurrency are as follows:

- **Non-atomic:** Non-atomic operations are interruptible by several processes that can lead to problems.
- **Race conditions:** It is an undesirable situation which occurs when a system tends to perform two or more operations at the same time. Thus, the operations need to be done in a sequence. It also occurs when output depends on multiple processes to gets a point first.
- **Blocking:** Blocking means any flow of the work is suddenly went into halting state. Processes can also block waiting for resources and it can block input for a long period. Periodically if any process required to update data then this will be very undesirable.
- **Starvation:** It occurs when a high priority of processes keeps executing and low priority processes get blocked for an indefinite time.
- **Deadlock:** It occurs when more than two processes are blocked and any of processes cannot proceed to execute.

8.3 Message Passing

Generally, the communication method for imperative programs depends on shared memory or message passing. In shared-memory programming, multiple threads are accessible for a program's variable to communicate for a pair of threads. Thus, anyone of them wants to write a value to a variable then it can simply read it. However, in message-passing programming, the threads have no common links to communicate with a pair of threads. Thus, to communicate one should perform an explicit and send operation to transmit data from one to another. The message passing model, as shown in Figure 1, can process communication between the threads.





It allows multiple processes to read and write data. The message queue can process to read and write data without being connected and it stores messages on the queue until the receiver does not receive. For inter-process communication, message queues are useful and many operating systems used this. In the mode both the processes, P1 and P2 can access the message queue and store and retrieve data. However, the message-passing model is easy to implement than the shared memory model. The parallel hardware can also be built by using message passing model because it has the ability to tolerant of higher communication latencies. Sometimes, it has slower communication than the shared memory model because the connection setup takes much time.

An approach based on message passing model the processes must operate in the isolation state. It should do not exchange information, when it is necessary then it should send and

It allows multiple processes to read and write data. The message queue can process to read and write data without being connected and it stores messages on the queue until the receiver does not receive. For inter-process communication, message queues are useful and many operating systems used this. In the mode both the processes, P1 and P2 can access the message queue and store and retrieve data. However, the message-passing model is easy to implement than the shared memory model. The parallel hardware can also be built by using message passing model because it has the ability to tolerant of higher communication latencies. Sometimes, it has slower communication than the shared memory model because the connection setup takes much time.

An approach based on message passing model the processes must operate in the isolation state. It should do not exchange information, when it is necessary then it should send and receive messages. The synchronisation between processes can be defined by the blocking method and can attach to the send and receive messages. The most common combinations are as follows:

- **Non-blocking send:** It continues to execute when a process sends a message without receiving a process.
- **Blocking receive:** It blocks the message until it is available or a process attempts to receive a message.

A set of processes to share a mailbox can be achieved by message passing with the blocking method, mutual exclusion. Initially, some of the messages are sent to box by the system, then the following code can be used to execute each process, whenever it wants to enter into critical section:

```
receive (box);

<critical code section>

send (box);
```

8.4 Multi-threaded Programs

A thread is the smallest unit of processing. Threads make our program faster as more than one thread can be executed simultaneously. They are executed independently, and if they are not synchronized, they can produce wrong output or create a deadlock condition.

Java is a multithreaded programming language. This feature of Java helps in making a Java program run in multiple threads. When we create multiple threads, the synchronisation of the threads should be strictly maintained. Multithreading provides the advantage of saving the memory as a common area is shared by the threads. In simpler words, multithreading feature of Java is used to achieve multitasking. This feature of Java is used for creating games.

8.4.1 Threads

Threads are a part of a process that can run independently. They are lightweight and run simultaneously. Multithreading is more suitable than multiprocessing to achieve multitasking as the threads share a common memory in multithreaded programs. On the other hand, in multiprocessing, each process has its own memory. Some of the important properties of threads are as follows:

- Threads are lightweight.
- Threads run independently.
- Threads of a process share the same memory location.
- Communication cost of the threads of a process is very low.

8.4.2 Thread Life Cycle

The life cycle of a thread includes the total processes that it performs from the start of a thread to its termination. Java Virtual Machine (JVM)controls the lifecycle of a

8.4.2 Thread Life Cycle

The life cycle of a thread includes the total processes that it performs from the start of a thread to its termination. Java Virtual Machine (JVM)controls the lifecycle of a thread in Java. Figure 2 illustrates the life cycle of a thread in Java:

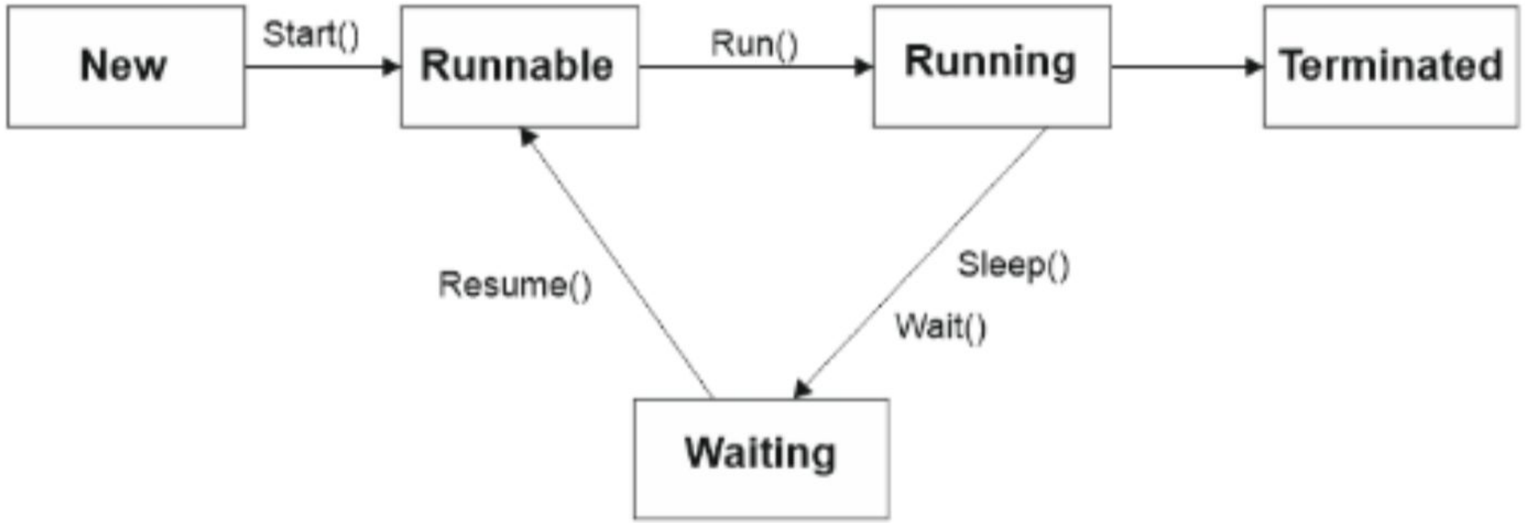


Figure 2: Life cycle of a Thread

Let us understand the different stages of a thread life cycle.

- **New**: A thread begins its life at the new state, where we create an instance of the Thread class. In this state, the thread is called a born thread. A thread is in this state until a process invokes the start() method.
- **Runnable**: When a process invokes the start() method on a newly born thread, the thread comes in the runnable state. In the runnable state, the thread is ready for execution.
- **Running**: When thread scheduler is selected, a thread for execution is called in running state.
- **Waiting**: When a thread needs some IO information, it goes in the waiting state. After the waiting state, the thread again moves to the runnable state.

Let us understand the different stages of a thread life cycle.

- **New**: A thread begins its life at the new state, where we create an instance of the Thread class. In this state, the thread is called a born thread. A thread is in this state until a process invokes the start() method.
- **Runnable**: When a process invokes the start() method on a newly born thread, the thread comes in the runnable state. In the runnable state, the thread is ready for execution.
- **Running**: When thread scheduler is selected, a thread for execution is called in running state.
- **Waiting**: When a thread needs some IO information, it goes in the waiting state. After the waiting state, the thread again moves to the runnable state.
- **Terminated**: A thread is in the terminated state if the task of the run()method is completed or terminated.

8.4.3 Methods of Thread Class

The Thread class provides different methods to create and perform different actions on a thread. Some of the common methods of the Thread class are listed in Table 2:

Table 2: Commonly used Methods of Thread Class

Method Name	**Description**
public void setName(String name)	Changes the name of the thread
public Thread currentThread()	Returns the reference of currently executing thread
public int getId()	Returns the ID of the thread
public Thread.State getState()	Returns the state of the thread

public Thread.State getState()	Returns the state of the thread
public boolean isAlive()	Tests if the thread is alive
public void yield()	Pauses the currently executing thread temporarily
public void interrupt()	Interrupts the thread
public boolean isInterrupted()	Tests if a thread has been interrupted
public static boolean interrupted()	Tests if the current thread has been interrupted

8.4.4 Thread Creation Syntax

A thread can be created in Java in two ways, either by extending the Thread class or by implementing the runnable interface. Let us learn about both of these.

8.4.5 By extending the Thread class

We can create a thread by extending the Thread class. When we extend the Thread class for creating a thread, we use the method of the Thread class for performing various actions on the thread. We can create a thread by performing the following two simple steps:

8.4.5 By extending the Thread class

We can create a thread by extending the Thread class. When we extend the Thread class for creating a thread, we use the method of the Thread class for performing various actions on the thread. We can create a thread by performing the following two simple steps:

1. For creating a thread, the first step is to override the run() method. This method is present in the Thread class. We write our logic or task that the thread will perform in the body of the run() method. The syntax is as follows:

```
Public void run() {

// body of run ;
```

•

2. When a thread object is created in our program, we call the start() method. The start() method in turn calls the run() method. The syntax is as follows:

```
void start();
```

In Example 1, we are illustrating how to create a thread by extending the Thread class. In this program, we are creating a thread called mtc.

Example 1: Creating a thread by extending Thread class

```
class ExampleCreatThread1 extends Thread{

    public void run(){

        System.out.println("Message of thread ..... ");

    }
```

```
    public void run(){

        System.out.println("Message of thread ..... ");

    }

    public static void main(String args[]){

        ExampleCreatThread1 mtc = new ExampleCreatThread1();

        mtc.start();

    }

}
```

Figure 3 shows the output of Example 1:

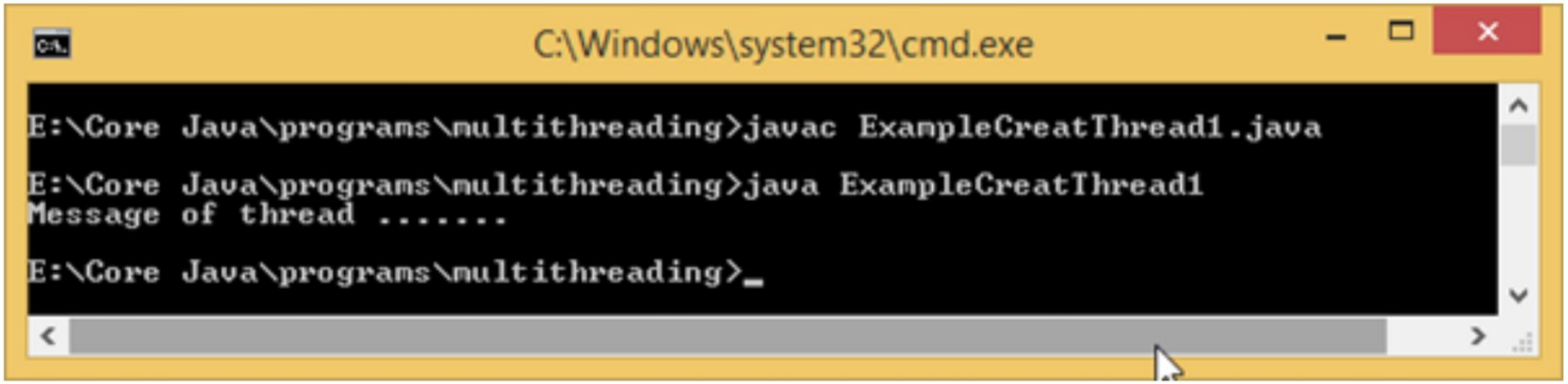


Figure 3: Output of Example 1

8.4.6 By Implementing the Runnable Interface

We can create a thread by implementing the runnable interface. For creating a thread by implementing the runnable interface, we need to perform the same step(as performed for extending the Thread class), i.e., overriding the run() method and calling the start() method. In Example 2, we are illustrating how to create a thread by implementing the runnable interface. In this program, we are creating a thread called mtc.

Example 2: Creating a thread by Implementing the Runnable Interface

```
class ExampleCreatThread2 implements Runnable{

    public void run(){

        System.out.println("Message of thread ..... ");

    }

    public static void main(String args[]){

        ExampleCreatThread2 mtc = new ExampleCreatThread2();

        Thread th = new Thread(mtc);

        th.start();

    }

}
```

Figure 4 shows the output of Example 2:

Figure 4 shows the output of Example 2:

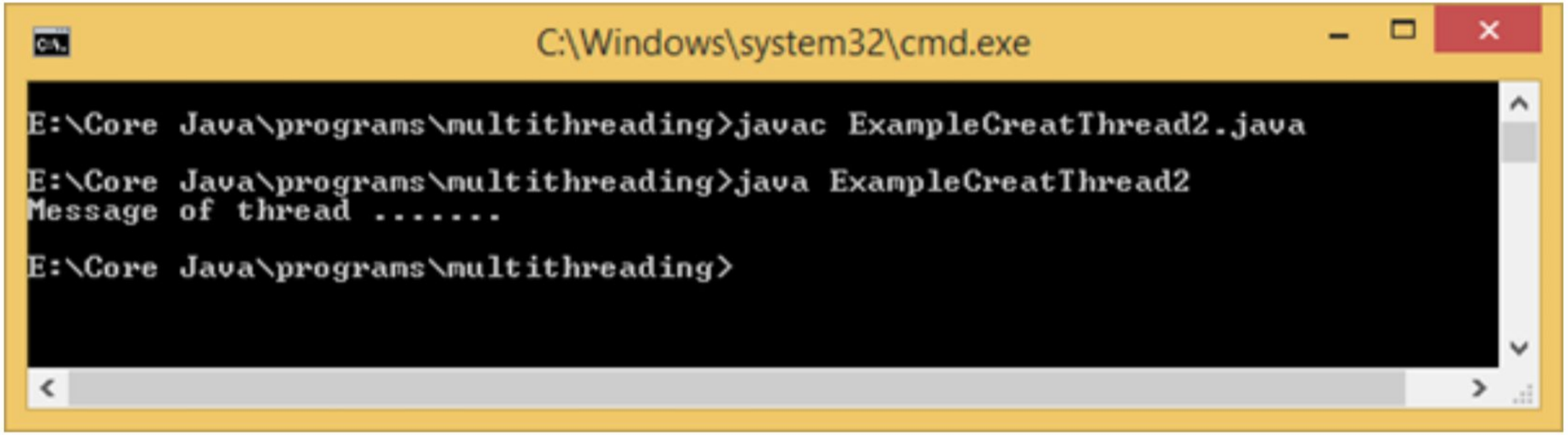


Figure 4: Output of Example 2

8.4.7 Creating Multiple Threads

In Examples 1 and 2, we have created a single thread. Similarly, we can create multiple threads. We can create multiple threads either by extending the Thread class or by implementing the runnable interface. For creating multiple threads, we just need to create the multiple thread object and call the start() method on the thread object.

In Example 3, we are illustrating how to create multiple threads:

Example 3: Creating Multiple Threads

public class ExampleCreatMultiThread extends Thread {

```
public class ExampleCreatMultiThread extends Thread {

public void run() {

for (int i = 1; i <= 5; i++) {

                    System.out.println(" Thread name = "+ Thread.currentThread().getName());

}

}

public static void main(String[] args) {

ExampleCreatMultiThread t1 = new ExampleCreatMultiThread();

t1.start();

ExampleCreatMultiThread t2 = new ExampleCreatMultiThread();

t2.start();

ExampleCreatMultiThread t3 = new ExampleCreatMultiThread();

t3.start();

}

}
```

In Program 3, we are extending a Thread class and creating a Thread class method “Thread.currentThread().getName())”.

In Program 3, we are extending a Thread class and creating a Thread class method “Thread.currentThread().getName())”.

Figure 5 shows the output of Example 3:

```
C:\Windows\system32\cmd.exe

E:\Core Java\programs\multithreading>javac ExampleCreatMultiThread.java

E:\Core Java\programs\multithreading>java ExampleCreatMultiThread
 Thread name = Thread-0
 Thread name = Thread-2
 Thread name = Thread-1
 Thread name = Thread-2
 Thread name = Thread-0
 Thread name = Thread-2
 Thread name = Thread-1
 Thread name = Thread-2
 Thread name = Thread-0
 Thread name = Thread-2
 Thread name = Thread-1
 Thread name = Thread-0
 Thread name = Thread-1
 Thread name = Thread-0
 Thread name = Thread-1

E:\Core Java\programs\multithreading>_
```

Figure 5: Output of Example 3

Figure 5: Output of Example 3

<H2> Thread Implementation using C++

In C++, std::thread is the thread class which is used to represents a single thread. To start a thread, a new thread object is created and the executing code to be called (i.e, a callable object) is passed into the constructor of the object. Once the object is created a new thread is launched which executes the code specified in callable.

A callable object can either be:

1. A function pointer
2. A function object
3. A lambda expression

After defining callable, it is passed to the constructor.

EXAMPLE: // C++ program to demonstrate multithreading using all three callables.

```
#include <iostream>

#include <thread>

using namespace std;



// Callable Function

void callable_func(int A)
```

```
// Callable Function
void callable_func(int A)
{
	for (int i = 0; i < A; i++) {
		cout << "Thread having function pointer as callable :"<< i<<"\n";
	}
}

// A callable object
class thread_obj {
public:
	void operator()(int B)
	{
		for (int i = 0; i < B; i++)
			cout << "Thread using callable function object :"<< i<<"\n";
	}
```

```
};


int main()
{
        cout << "Threads 1 and 2 and 3 operating independently" << endl;


        // This thread is launched by using callable function pointer
        thread th1(callable_func, 4);


        // This thread is launched by using callable function object
        thread th2(thread_obj(), 3);


        // A Lambda Expression
        auto l = [](int C) {
                for (int i = 0; i < C; i++)
                        cout << "Thread using lambda expression as callable :"<< i<<"\n";  };
```

```
    // This thread is launched by using lambda expression as callable

    thread th3(l, 4);


    // Waiting for the threads to finish

    // Wait for thread t1 to finish

    th1.join();


    // Wait for thread t2 to finish

    th2.join();


    // Wait for thread t3 to finish

    th3.join();


    return 0;

}
```

Executing the code can produce a similar output:

Executing the code can produce a similar output:

Threads 1 and 2 and 3 operating independently

Thread having function pointer as callable :0

Thread having function pointer as callable :1

Thread using callable function object :0

Thread using lambda expression as callable :0

Thread having function pointer as callable :2

Thread having function pointer as callable :3

Thread using callable function object :1

Thread using lambda expression as callable :1

Thread using callable function object :2

Thread using lambda expression as callable :2

Thread using lambda expression as callable :3

8.5 Implementing Synchronization

Synchronization is necessary when we create a multithreaded program as the threads are accessing a shared resource. If the multiple threads access a shared resource in an unsynchronized manner, then the desired output is not achieved. For understanding synchronization, let us first check what happens if we do not use synchronization.

In Example 4,we are illustrating a multithreaded program that is not synchronized:

Example 4: An Unsynchronized Program

```
class Character{
    void printCharacter(char n){
        for(int i=1;i<6;i++){
            System.out.println(n);
            try{
                Thread.sleep(500);
            }
            catch(Exception ex){
                System.out.println(ex);
            }
```

```
            }
        }
    }
}
class Thread1 extends Thread{
    Character c;
    Thread1(Character c){
        this.c=c;
    }
    public void run(){
        c.printCharacter('*');
    }
}
class Thread2 extends Thread{
    Character c;
    Thread2(Character c){
```

```
}
class Thread2 extends Thread{
    Character c;
    Thread2(Character c){
        this.c=c;
    }
    public void run(){
        c.printCharacter('#');
    }
}
class ExampleSynchronization{
    public static void main(String args[]){
        Character obj = new Character();
        Thread1 t1=new Thread1(obj);
        Thread2 t2=new Thread2(obj);
```

```
        t1.start();

        t2.start();

    }

}
```

In Program 4, we are trying to print two characters * and # in a sequence. The desired output is to first print the * character five times and then print the # character five times.

Figure 6 shows the output of Example 4:

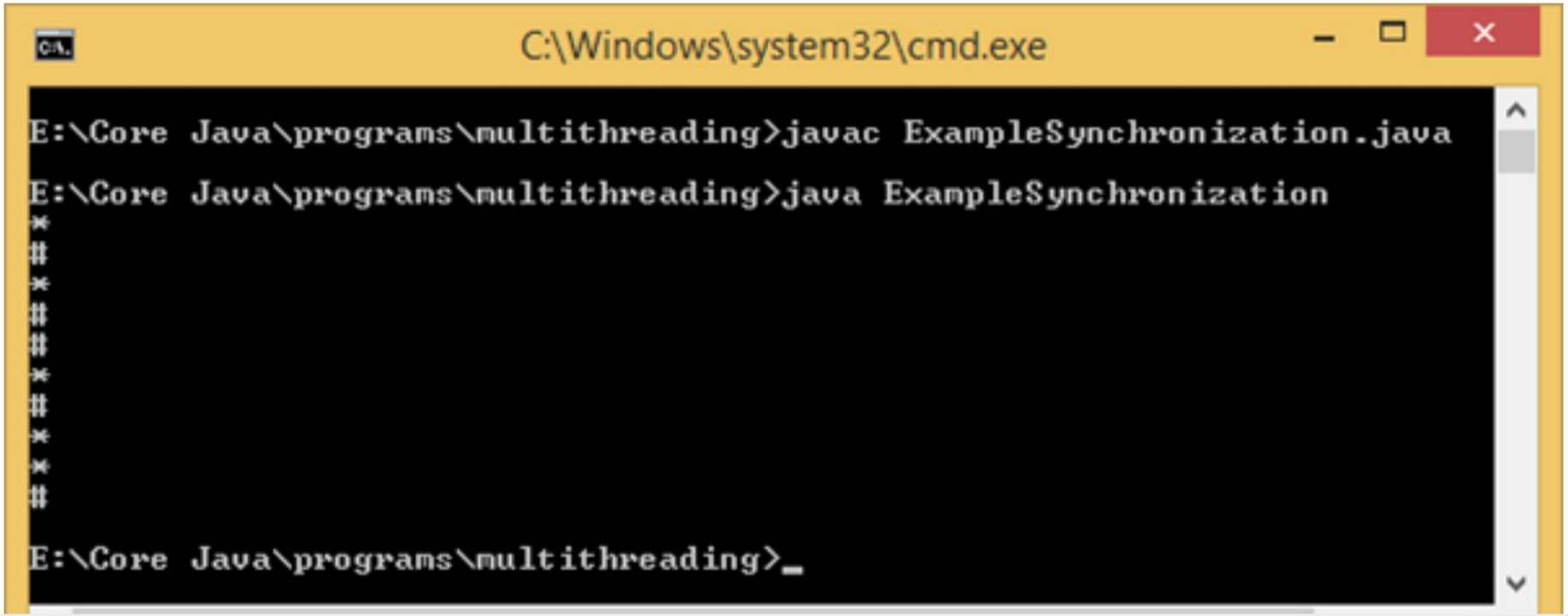


Figure 6: Output of Example 4

As you can see in Figure 6, the characters are not printed as specified. For removing this problem, Java provides the synchronized keyword. This keyword helps in creating either a synchronized method or a synchronized block to handle synchronization. When a thread calls the synchronized method, the method acquires a lock for that thread and releases the lock if the thread completes the task.

In Example 5, we are illustrating the use of the synchronized keyword for executing multiple threads in a synchronized manner. In this program, we create the print Character() method synchronized to achieve the desired result.

Example 5: A Synchronized Program

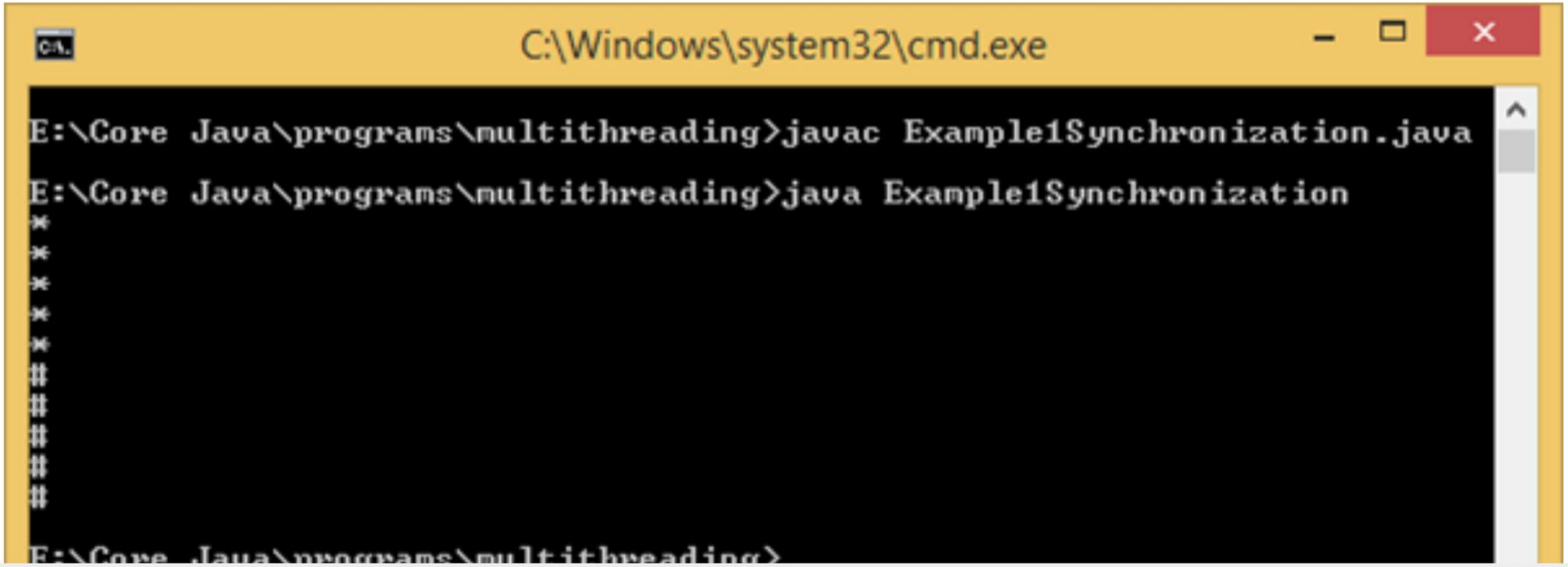
```
class Character{

    synchronized void printCharacter(char n){

        for(int i=1;i<6;i++){

            System.out.println(n);

            try{

                Thread.sleep(500);

            }

            catch(Exception ex){

                System.out.println(ex);

            }
```

```
                System.out.println(ex);
            }
        }
    }
}
class Thread1 extends Thread{
    Character c;
    Thread1(Character c){
        this.c=c;
    }
    public void run(){
        c.printCharacter('*');
    }
}
class Thread2 extends Thread{
    Character c;
```

```
class Thread2 extends Thread{
    Character c;
    Thread2(Character c){
        this.c=c;
    }
    public void run(){
        c.printCharacter('#');
    }
}
class Example1Synchronization{
    public static void main(String args[]){
        Character obj = new Character();
        Thread1 t1=new Thread1(obj);
        Thread2 t2=new Thread2(obj);
        t1.start();
        t2.start();
```

```
        Thread1 t1=new Thread1(obj);

        Thread2 t2=new Thread2(obj);

        t1.start();

        t2.start();

    }

}
```

Figure 6 shows the output of Example 5:



8.6 Deadlock

In a multithreaded program, a deadlock is a situation in which all the threads are waiting for a locked resource. This condition occurs in a Java program due to unordered use of the synchronized keyword. Figure 8 illustrates the deadlock condition:

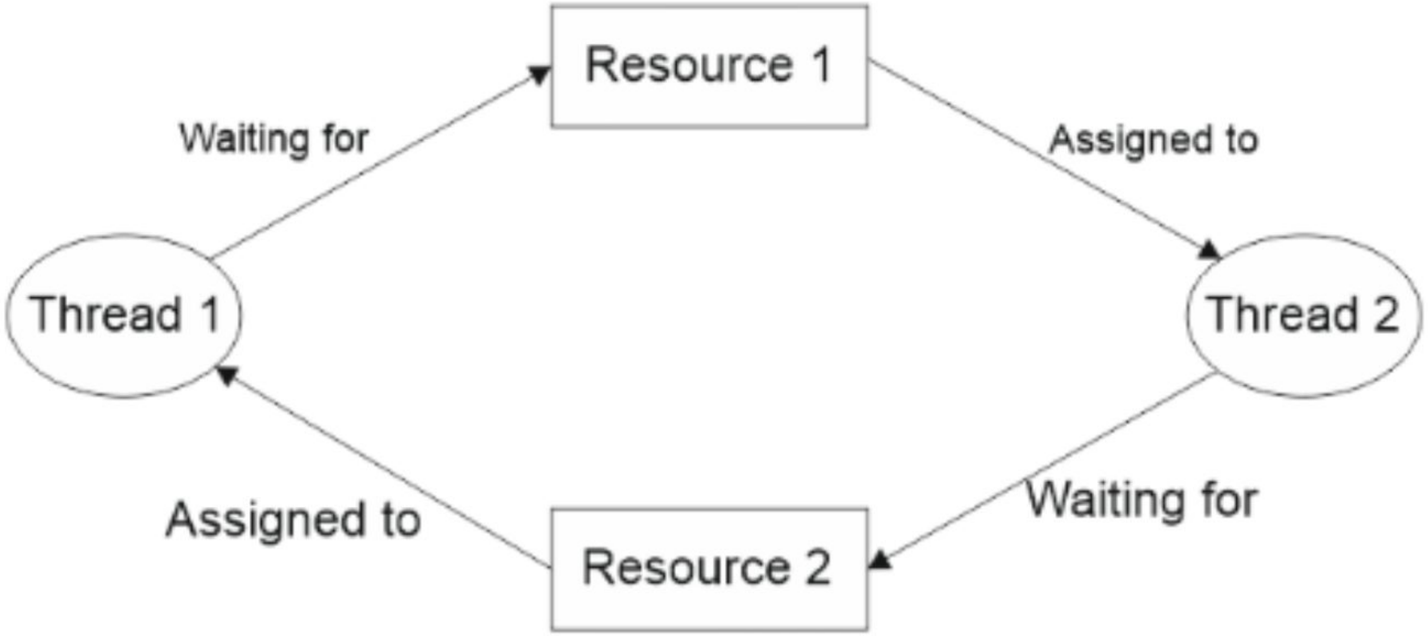


Figure 7: Deadlock condition

Figure 8: Deadlock condition

Figure 8: Deadlock condition

Figure 8 shows two resources: Resource 1 and Resource 2, and two threads: Thread 1 and Thread 2. Thread 1 is waiting for resource 1 while acquiring resource 2, and thread 2 is waiting for resource 2 while acquiring resource 1. Both the threads are in waiting condition. This is called a deadlock condition.

In Example 6, we are illustrating a deadlock situation created by two threads. In this program, deadlock occurs because of using the synchronized keyword in a wrong order.

Example 6: A Deadlock Situation

```
public class ExampleDeadlock {

    public static void main(String[] args){

        final String res1 = "First resource";

        final String res2 = "Second resource";



        Thread t1 = new Thread() {

            public void run() {

                synchronized(res1){

                    try{

                        Thread.sleep(100);

                    }
```

```
                Thread.sleep(100);
            }
            catch (InterruptedException ex){
            }
            synchronized(res2){
                System.out.println("Second resource is locked by first thread");
            }
        }
    }
};

Thread t2 = new Thread(){
    public void run(){
        synchronized(res2){
            System.out.println(" Second resource is locked by second thread");
```

```
Thread t2 = new Thread(){
    public void run(){
        synchronized(res2){
            System.out.println(" Second resource is locked by second thread");
            try{
                Thread.sleep(100);
            }
            catch (InterruptedException ex){
            }
            synchronized(res1){
                System.out.println("First resource is locked by second thread");
            }
        }
    }
};
```

```
        t1.start();

        t2.start();

    }

}
```

Figure 9 shows the output of Example 6:

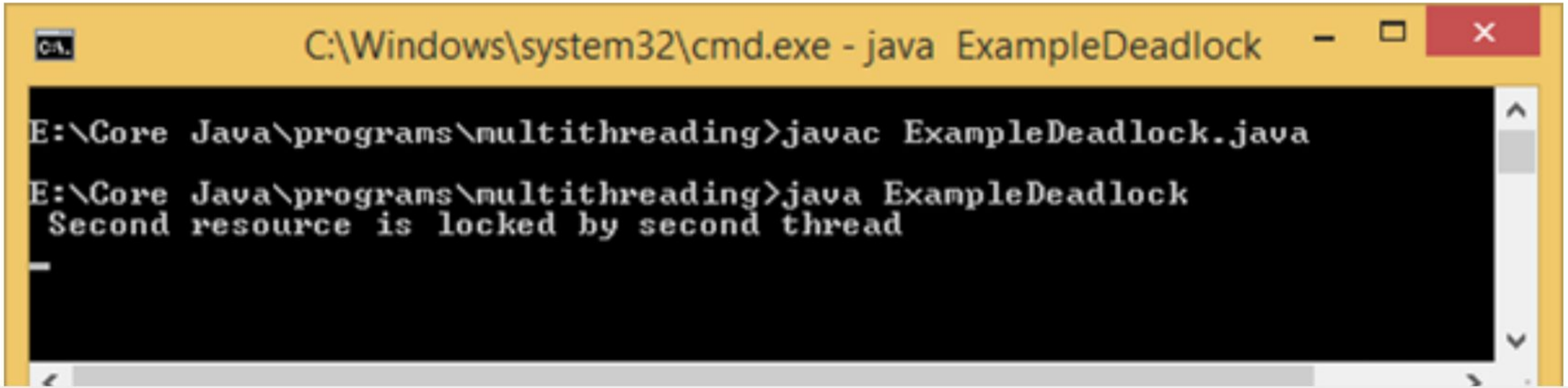


Figure 9: Output of Example 6

As you can see in Figure 8, the output of Program 6 goes in a deadlock situation and the output screen hangs. In such a situation, press the Ctrl + C keys to come out of the program.

In Example 7,we are solving this deadlock situation by just providing the right order of the synchronized keyword:

Example 7: Solution of deadlock

```
public class ExampleDeadlockRemov {

    public static void main(String[] args){

        final String res1 = "First resource";

        final String res2 = "Second resource";



        Thread t1 = new Thread() {

            public void run() {

                synchronized(res1){

                    System.out.println("First resource is locked by first thread");

                    try{

                        Thread.sleep(100);
```

```
            try{
                Thread.sleep(100);
            }
            catch (InterruptedException ex){
            }
            synchronized(res2){
                System.out.println("Second resource is locked by first thread");
            }
        }
    }
};

Thread t2 = new Thread(){
    public void run(){
        synchronized(res1){
            System.out.println(" First Second resource is locked by second thread");
```

```
        synchronized(res1){
            System.out.println(" First Second resource is locked by second thread");
            try{
                Thread.sleep(100);
            }
            catch (InterruptedException ex){
            }
            synchronized(res2){
                System.out.println("Second resource is locked by second thread");
            }
        }
    }
};

t1.start();
t2.start();
```

```
        t1.start();

        t2.start();

    }

}
```

Figure 10 shows the output of Example 7:

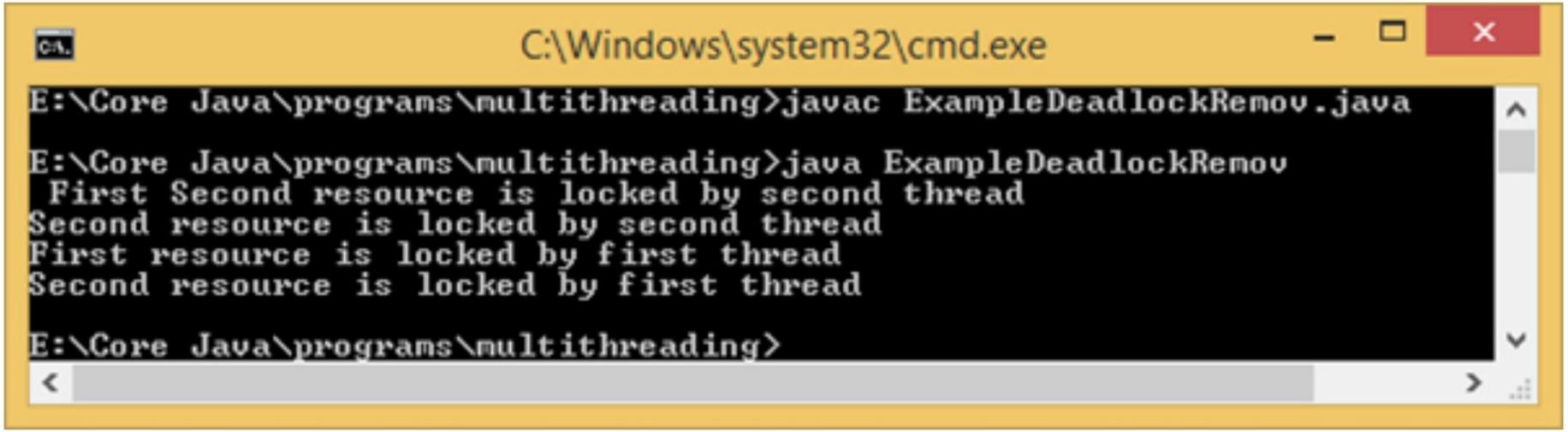


Figure 10: Output of Example 7

8.7 Interthread Communication and Synchronization

Interthread communication is the process in which two synchronized threads are communicating with each other. It can be performed by using the following three methods of the object class: wait(), notify(), notifyAll().

Let us learn about the three methods of the object class.

- **wait()**: When the wait() method is called, it causes the current thread to move in the waiting condition until another thread invokes the notify() and notifyAll() methods. The syntax is as follows:

public final void wait();

- **notify()**: When the notify()method is called, it wakes up a single thread of the calling object. The syntax is as follows:

public final void notify();

- **notifyAll()**: When the notifyAll() method is called, it wakes up all the threads of the calling object. The syntax is as follows:

public final void notifyAll();

The sleep() method of the Thread class is used to make a thread sleep for a specified amount of time. The differences between the wait()and sleep() methods are listed in Table 3:

Table 3: Difference Between wait() and sleep() Methods

wait()	sleep()
This method belongs to the Object class.	This method belongs to the Thread class.

wait()	sleep()
This method belongs to the Object class.	This method belongs to the Thread class.
This is a non-static method.	This is a static method.
This method releases lock.	This method does not release lock.
This method should be notified by either notify() or notifyAll().	This method is completed after a specified amount of time.

In Example 8, we are illustrating the interthread communication:

Example 8: Interthread Communication

```
class Communicate {
  boolean flag = false;


  public synchronized void FirstPerson(String message) {
    if (flag) {
```

```
public synchronized void FirstPerson(String message) {
  if (flag) {
    try {
      wait();
    }catch (InterruptedException ex) {
      ex.printStackTrace();
    }
  }
  System.out.println(message);
  flag = true;
  notify();
}

public synchronized void SecondPerson(String message) {
  if (!flag) {
    try {
```

```
        try {
          wait();
        }catch (InterruptedException ex) {
          ex.printStackTrace();
        }
      }

      System.out.println(message);
      flag = false;
      notify();
    }
  }

class P1 implements Runnable {
  Communicate m;
  String[] str1 = { "Hello", "How are you ?", "i am also fine!" };
```

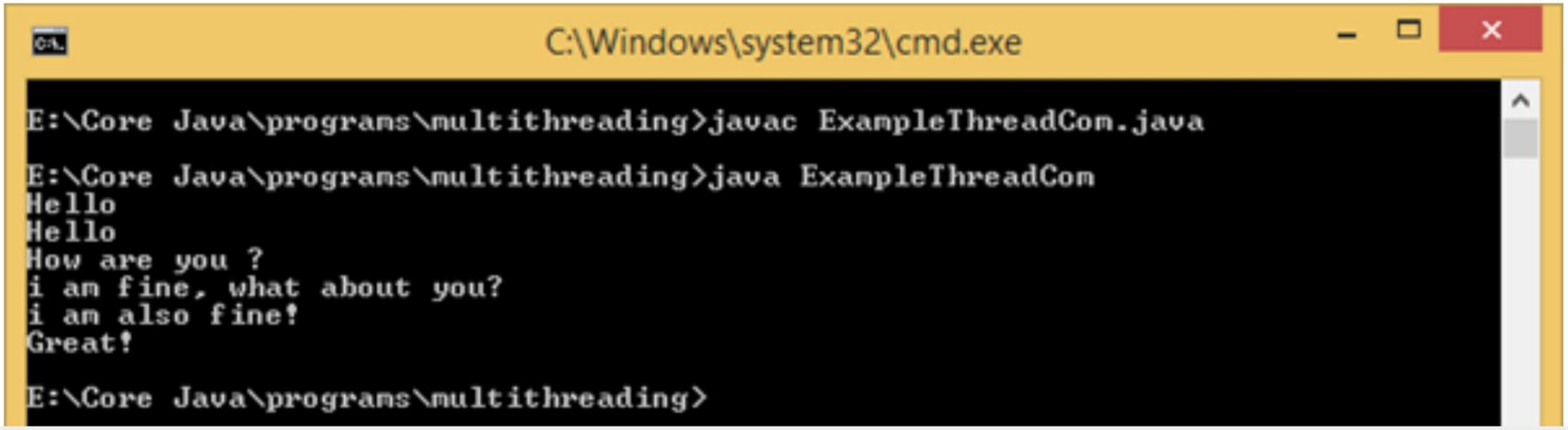
```
  String[] str1 = { "Hello", "How are you ?", "i am also fine!" };


  public P1(Communicate m1) {

    this.m = m1;

    new Thread(this, "FirstPerson").start();

  }


  public void run() {

    for (int i = 0; i < str1.length; i++) {

      m.FirstPerson(str1[i]);

    }

  }

}


class P2 implements Runnable {
```

```
class P2 implements Runnable {
  Communicate m;
  String[] str2 = { "Hello", "i am fine, what about you?", "Great!" };

  public P2(Communicate m2) {
    this.m = m2;
    new Thread(this, "SecondPerson").start();
  }

  public void run() {
    for (int i = 0; i < str2.length; i++) {
      m.SecondPerson(str2[i]);
    }
  }
}
public class ExampleThreadCom {
```

```
public class ExampleThreadCom {

  public static void main(String[] args) {

    Communicate m = new Communicate();

    new P1(m);

    new P2(m);

  }

}
```

In this program, we are creating two threads P1 and P2 and showing communication between these threads.

Figure 11 shows the output of Example 8:



8.8 Language and Libraries

Thread-level concurrency can be included using the form of explicitly concurrent languages, compiler-supported extensions to traditional sequential languages, or library packages outside the language proper. These three options are widely used where the shared-memory languages are used for multicore and small multiprocessor machines, and message-passing libraries are used for massively parallel super-computers. Table 4 categorises these systems:

	Shared memory	**Message passing**	**Distributed computing**
Language	Java, C#		
Extension	OpenMP		Remote procedure call
Library	pthreads, Win32 threads	MPI	Internet libraries

For various years, most parallel programming use sequential languages that has libraries for synchronization or message passing. Its still being used today. In Unix, the shared memory parallelism uses the POSIX pthreads standard, which includes mechanisms to create, destroy, schedule, and synchronize threads.

8.9 Recap and Questions

8.9.1 Recap

Let us quickly recapitulate the following discussion points:

- In programming concurrent refers to computing model in which more than one processors at a time can execute instructions simultaneously for better performance.
- A thread is the smallest unit of processing.
- Java is a multithreaded programming language.
- Synchronization is necessary when we create a multithreaded program as the threads are accessing a shared resource.
- In a multithreaded program, a deadlock is a situation in which all the threads are waiting for a locked resource.
-

8.9.2 Test Your Skills- Multiple Choice Questions

1. Which of the following exceptions is thrown by the sleep method?

a. ThreadException
b. InterruptedException
c. NullPointerException
d. IOException

2. Which of the following methods is used to create multiple threads?

2. Which of the following methods is used to create multiple threads?

a. kill()
b. start()
c. stop()
d. begin()

3. Alive method is used for which of the following?

a. To know whether a thread is running or not
b. To create a thread after stopping it
c. To know whether a thread was created or not
d. To resume a thread after blocking it

4. Which of the following is a Thread class method?

a. wait()
b. run()
c. notify()
d. notifyAll()

5. How many stages are there in the thread life cycle?

a. 3
b. 4
c. 5
d. 6

c. 5
d. 6

8.9.3 Fill in the Blanks

1. In Java, a thread can be created either by _____ or _____.
2. When we create a thread, the _____ method of the Thread class overrides.
3. The start() method belongs to the _____ Class.
4. The _____ method is used to resume a thread.
5. The notify() method belongs to the _____ Class.

8.9.4 Theoretical Questions

1. What is deadlock?
2. What would happen if we do not use synchronization in a multithreaded program?
3. Discuss the thread life cycle.
4. Describe various methods to create a thread.
5. What do you mean by interthread communication?

Chapter 9 :Scripting Languages

Learning objectives

At the end of this module, you will be able to:

- Describe scripting language
- Describe different problem domains for using scripting
- Discuss the use of scripting in web development
- Explain innovative features of scripting language

9.1 Introduction

A scripting language is a computer language with a series of inbuilt commands. The basic difference between scripting language and programming language is security. Scripting languages are more secure and do not have capability of system programming making them secure. Another major difference is, scripting languages are interpreted while programming languages require special compilers for execution. It does not require the compilation step for a program but rather are interpreted. For example, a C program needs to be compiled before running whereas a scripting language like JavaScript or PHP does not need to be compiled. It became globally because of the development of the Internet as a communications tool. It automates the execution of tasks and the tasks are executed alternatively one-by-one by a human operator. Scripting languages provide interpreted rather than compiled. It also ranges from low to high domain-specific languages to general-purpose programming languages for scripting.

Typically, scripting languages learn and write code fast, even a short source code files or interactively in a read–eval–print loop language shell. It indicates that simple syntax and

- Describe scripting language
- Describe different problem domains for using scripting
- Discuss the use of scripting in web development
- Explain innovative features of scripting language

9.1 Introduction

A scripting language is a computer language with a series of inbuilt commands. The basic difference between scripting language and programming language is security. Scripting languages are more secure and do not have capability of system programming making them secure. Another major difference is, scripting languages are interpreted while programming languages require special compilers for execution. It does not require the compilation step for a program but rather are interpreted. For example, a C program needs to be compiled before running whereas a scripting language like JavaScript or PHP does not need to be compiled. It became globally because of the development of the Internet as a communications tool. It automates the execution of tasks and the tasks are executed alternatively one-by-one by a human operator. Scripting languages provide interpreted rather than compiled. It also ranges from low to high domain-specific languages to general-purpose programming languages for scripting.

Typically, scripting languages learn and write code fast, even a short source code files or interactively in a read–eval–print loop language shell. It indicates that simple syntax and semantics of a script is executed from start to finish without any entry point.

In this chapter, you will learn about the term scripting language and its common characteristics. You will get familiar with different problem domains for using scripting. You will learn about the use of scripting in web development. At the end of the chapter, you will learn about the innovative features of scripting languages.

9.2 Common Characteristics

A scripting language is an interpreted programming language in which it translates code into the machine. It is made for specific runtime environments where it automates some of the code which does not require the compilation step. It also produces some new functions for applications and complex system. Thus, it is designed for integrating and communicating with other programming languages. For example, JavaScript, Python, and Ruby and so on. Generally, there are two types of scripting are as follows:

- **Server-side scripting languages:** It runs on a web server. Where a client sends a request to the server and it responds via HTTP.
- **Client-side scripting languages:** It runs on the client end on their web browser. It reduces the demand on the server by allowing web pages to load faster.

There are some common characteristics in scripting language are as follows:

- Cost effective
- Lack of declarations
- Dynamic typing
- System flexibilty
- High-level data types

High-level data types

Figure 1: Common Characteristics of Scripting Language

- **Interactive:** A scripting language like in ruby has a compiler which focuses on reading source program before producing an output. Ruby accept all commands from the keyboard. However, many other languages like Rexx, Python, Tcl, compile their input line by line.
- **Cost-effective:** Scripting languages requires a smaller number of boilerplate to support rapid development and interactive use. Few languages have more punctuation and fewer identifiers like in Perl. While other programming languages like Rexx, Tcl, and AppleScript tend to have more English words and less punctuation. Thus, all these attempts are required to avoid the extensive declarations and top-level structure common to conventional languages.
- **Lack of declarations:** Most of the scripting languages have declarations to provide simple rules to govern the scope of names. For example, in Perl programming language. By default, it has everything. It also provides an optional declaration to limit a variable for a nested scope. In other languages like PHP and Tcl, whereby default everything is local and for global must be explicitly imported. Python has a different rule that is any variable which is assigned a value is local in which the assignment process appears. Where a special syntax is required to assign to a variable in a surrounding scope.

  For example, in Javascript we can directly use the variable without declaration.

  ```
  name='Jhon Scott';

  cnt=10;

  document.write(cnt);
  ```

  So here without actually declaring the variable , one can directly use it.

- **Dynamic typing:** Most scripting languages are dynamically typed in keeping with the lack of declarations. For example, PHP, Python, Ruby, and Scheme the typed variable will be checked immediately before use. Unlike other Rexx, Perl, and Tcl in which a variable will be interpreted differently.

```
document.write(cnt);
```

So here without actually declaring the variable , one can directly use it.

- **Dynamic typing:** Most scripting languages are dynamically typed in keeping with the lack of declarations. For example, PHP, Python, Ruby, and Scheme the typed variable will be checked immediately before use. Unlike other Rexx, Perl, and Tcl in which a variable will be interpreted differently.

  For example,

```
name= 'John Scott"

document.write("Name is " +name)  //here automatic concatenation is done
```

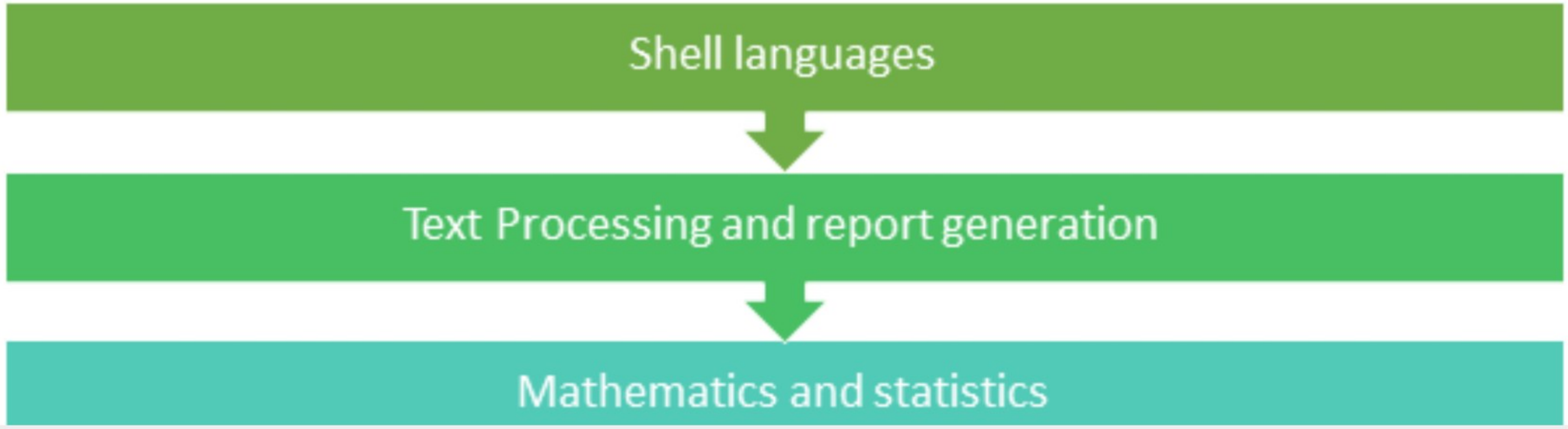
- **System flexibility:** Most programming languages provide a way for the operating system to run another program or to some perform operation directly. However, in scripting languages requests have direct support. For example, Perl provides many built-in commands which are easily accessed by the operating system functions. The built-in commands are as follows input and output, file and directory manipulation, process management, database access, sockets, inter-process communication, synchronisation, protection and authorization, time-of-day clock and network communication. Thus, these built-in commands are easier to use than corresponding library calls in languages like C.

High-level data types: The high-level data types such as sets, bags, dictionaries, lists, and tuples. These are common in the standard library packages of conventional programming languages. For example, the C++ language provides users to redefine standard infix operators. These are to make types as easy to use as more primitive and hardware-centric types. On another side, most scripting languages have an array common standard which is indexed by character strings, with an underlying implementation based on hash tables

9.3 Different Problem Domains for Using Scripting

Problem domain refers to all information which defines the problem and solution. It helps the owner to achieve the problem and context which exists in the problem. It also describes rules for essential functions of a solution product. It also represents the environment in which a solution will have to operate as well as the problem itself. It is also known as the scope of analysis. Where the user collects stories or requirements. For example, Scheme and Visual languages are widely used for scripting. Other languages like Perl, Python, and Ruby have their design for general-purpose use with features which can support large programs including modules, separate compilation, reflection, program development environments and so on. However, scripting languages have their principal use to define problem domains. In the days of punch-card computing, the user was allowed to script processing of a card deck by simple command languages like at the front of the deck a control card may indicate that the upcoming cards represented a program which needs to be compiled or an input program already has been compiled and stored in a database. Later, a control card contains the most recently executed program status and choose what to do next based on the program executed successfully.

In addition to design features for interactive use, the shell languages provide a method which can manipulate filenames, arguments, and commands, and so on. Most of these features are kept for general scripting languages and some of them use bash syntax. The different types of problem domain for using scripting are shown in Figure 2:



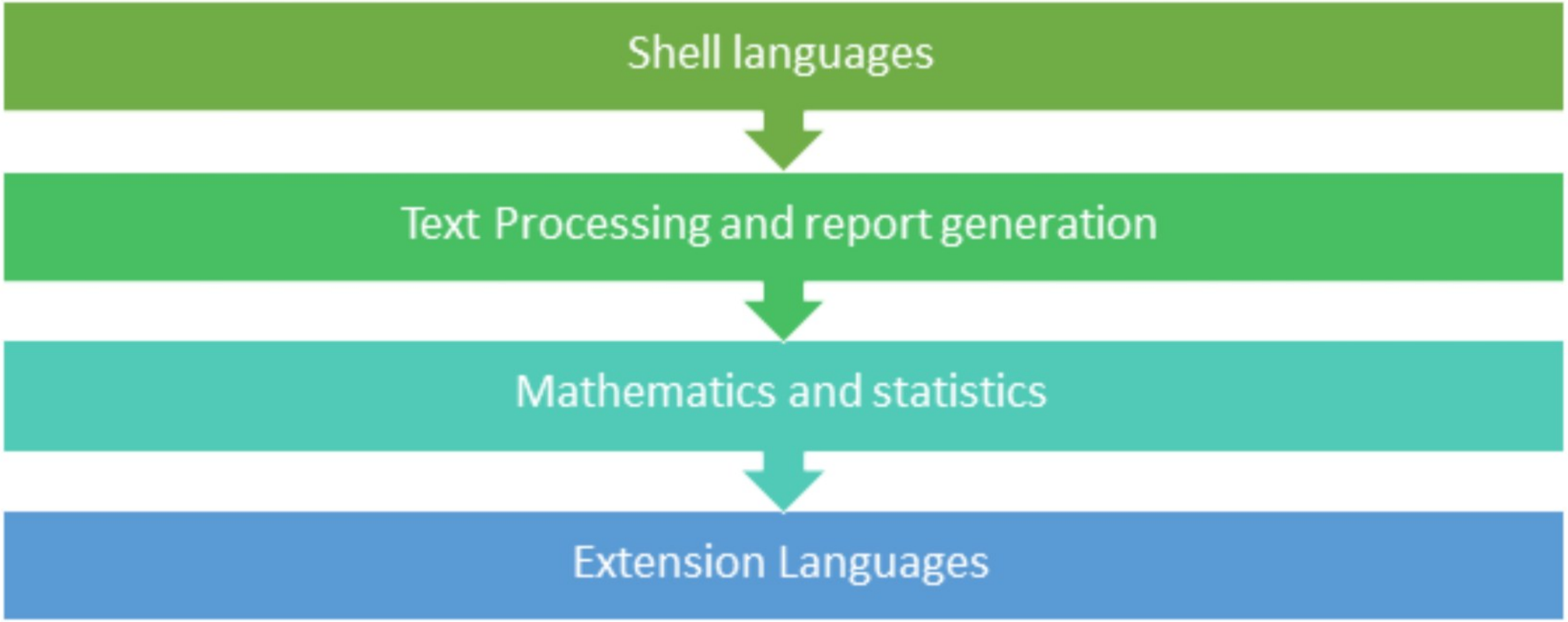


Figure 2: Different types of problem domains

- **Shell languages:** It have designed features for interactive use like Multics ~1964, Unix ~1973, sh, csh, tcsh, ksh, bash. It also provides a mechanism to manipulate file names, arguments, and commands, and to glue together other programs. Most of these features are retained by more general scripting languages. Generally, a mechanism is supported by Filename and Variable Expansion, Tests, Queries, and Conditions, Pipes and Redirection, Quoting and Expansion, Functions and the #! Convention.
- **Text Processing and report generation:** It provides no variables and no state it's a powerful filter. It processes one input line at a time. The first matching command is executed by "s/_/_/" substitution command.
- **Mathematics and statistics:** Scripting languages are oriented towards scientific and engineering applications. In mathematics, it provides support for numerical methods, symbolic mathematics, data visualization, mathematical modelling. Whereas in statistics its support multidimensional like arrays and lists, array slice ops, call-by-need, first-class functions, unlimited extent

- **Text Processing and report generation:** It provides no variables and no state it's a powerful filter. It processes one input line at a time. The first matching command is executed by "s/_/_/" substitution command.
- **Mathematics and statistics:** Scripting languages are oriented towards scientific and engineering applications. In mathematics, it provides support for numerical methods, symbolic mathematics, data visualization, mathematical modelling. Whereas in statistics its support multidimensional like arrays and lists, array slice ops, call-by-need, first-class functions, unlimited extent.
- **Extension Languages:** It serves to increase the usefulness of an application by allowing the user to create new commands, generally using the existing commands as primitives. It is an essential feature of tools like Adobe's graphics suite, JavaScript, Visual Basic or AppleScript.

9.3.1 Shell (Scripting) Language

The shell programming was made popular when Steve Bourne wrote the 'Bourne shell' ; it appeared in the Seventh Edition Bell Labs Research version of Unix. After that Bill Joy developed the 'C Shell' (tch). Other versions of shell are also existing, but most popular are these two. Basically shell programming, popularly known as shell scripting, are discovered to automate the chain of repetitive command line tasks. But these scripts not only execute the commands one after another but are basically a small programs. Scripting allows use of programming constructs while executing the commands. In itself, shell scripting is programming with commands to achieve the series of command line tasks like taking backup of files by appending the date at the end of filename. There are tedious tasks of executing the commands then just go for scripting it and simply with only one command.

Shell scripting allowed use of programming constructs directly into operating system interface which increased its use tremendously, particularly in the UNIX based operating system.

Lets write a simple script to print a String on to command line.

Example 9.1

```
#!/bin/bash

echo 'Hello World'  #This is a comment
```

When you run this on command line it will simply print 'Hello World'. You can declare variable also and use constructs like looping using for and conditions using if

When you run this on command line it will simply print ‘Hello World’. You can declare variable also and use constructs like looping using for and conditions using if.

Example 9.2

```
#!/bin/bash

msg=”This is first variable”

echo $msg
```

Above program will simply print the value of variable msg.

Example 9.3

```
#!/bin/bash

for i in 1 2 3 4 5

do

  echo "Looping ... number $i"

done
```

Above script will display:

```
Looping .... number 1

Looping .... number 2

Looping .... number 3
```

Looping .... number 3

Looping .... number 4

Looping .... number 5

The logical testing is also possible in shell scripting using if. The general syntax is :

```
if [ ... ]

then

  # if-code

else

  # else-code

fi
```

Example 9.4:

```
#!/bin/bash

X=4

if [ "$X" -lt "0" ]

then

  echo "X is less than zero"
```

```
fi
```

Example 9.4:

```
#!/bin/bash

X=4

if [ "$X" -lt "0" ]

then

  echo "X is less than zero"

else

  echo “X is not less than zero”

fi
```

Try above code and you will see the output as “X is not less than zero”. You may change the value and see the output.

There are many more constructs and concepts in shell scripting. Here we have just seen how shell scripting is useful in Operating System with command line interface.

9.4 Use of scripting in Web development

A scripting language is a programming language but with limited programming features. For example, Perl, Python, Javascript and other languages used for scripting are interpreted and does not require the compilation step. It also promotes the open project and used for web applications. It required less coding and allow the user to create and communicate with a website and provide flexibility, rapid development and dynamic checking. The type systems embrace very high-level concepts such as tables, patterns, lists and files. Languages such as Perl and python are known as glue language because these were designed to glue existing programs together. Some scripting languages are used in the process of web development.

Scripting languages are becoming more popular due to the emergence of web-based applications. Nowadays, the market is expanding for producing dynamic web content. The new scripting languages allow users with less or no programming programmer to develop interactive web pages with less effort. Over the past few years it has increased computer performance and unlike conventional programming languages still have some built-in security features.

9.4.1 Server side and Client side scripting

The scripts can be of two types depending on the use as : server side scripting or client side scripting. As the name indicates, in a typical client server application, the server side scripts are run and used at server while client side scripts runs in the browser of the client. Obviously client side scripts are useful for local calculations and display of the data. Client side script do not interact with server while running the script. The server side scripts run on the web server and process data at the web server.

Server side scripting languages can use more features for processing of data. Connecting with database, rendering of the GUI, complex data calculations are some of the important tasks performed at server side. Server side scripting is the communication medium between client and server. Earlier CGI (Common Gateway Interface) script was very popular to write the server side script. Now many options are available right from Javascript, VBScript, PHPScript etc.

Following is the simplest example of PHP script.

Example 9.5

```
<html>

<head><title>First php page</title></head>

<body>

<?php echo "<h1>Welcome to PHP</h1>";

$name="Deepali Vora";

echo "<h1 style=\"color:blue\"> $name </h1>" ;

?>

</body>

</html>
```

Here, special symbols <? Is used to start and ?> is used to end the script block. The script is embedded in Simple HTML page and is executed at the server side and then the output is sent to the client for the display. Client will not be able to see the script but will see only html part.

Following is a simple PHP server script code. It just illustrates the use of control statements in the server side script:

Example 9.6

```
<?php

//Print the HTML using the script
```

```
print <<<HTMLBLOCK

<html>

<head><title>Control structures</title></head>

<body>

HTMLBLOCK;


$i=0;

for($i=0; $i<5;$i++)    //Simple for loop

        print $i."<br>";

$a=array("red","blue","yellow");

foreach($a as $v)       // for loop used with an array

        print $v."<br>";


print <<<HTMLBLOCK

</body>

</html>
```

```
HTMLBLOCK;

?>
```

This is an example PHP script where the HTML tags are also printed using PHP Script language. The variable declaration is optional in scripting language here.

There are any simple functions which can be done at the client side before sending the request to the server to do the processing. Validation of input given by user is the best example of this. For example, while filling the registration form the text field which accepts username should not be blank. This checking can be done at client side browser only and is done using client side scripting. JavaScript is most prominently used client side scripting language and is supported by all browsers.

Following is the simple example of client side script which prints the today date.

Example 9.7

```
<!doctype html>

<html>

<head><title>Todays Date</title>

<script>

        var d=new Date();

        document.write(d.toLocaleDateString());

</script>

</head>
```

```
</head>

<body>

<h1>Todays Date</h1>

</body>

</html>
```

The client side script is included in the <script> tag of the HTML. Following is another simple example if form validation using JavaScript.

Example 9.8

```
<html>

<head><title>Registration Form</title>

<script type="text/javascript">

 function isBlank(x)

{

if(x.length==0)

return true;

var i=0;

for(i=0;i<x.length;i++)
```

```
    {
if(x.charCodeAt(i)!=95)
                return false;
else
return true;
    }
}
function valid()
{
if(isBlank(document.f1.t1.value))
        {
        alert("Enter your name");
        return false;
        }
return true;
}
```

```
}

</script></head>

<body bgcolor="#CE9B9B">

<h1>Registration Form</h1>

<form name="f1" method="post" align="center">

<table align="left" border="0">

<tr><td><b>Full Name:</b></td>

<td><input type="text" name="t1"/><br></td></tr>

</tr><td align="center"><input type="submit" name="submit" onClick="return valid();" value="Submit" /></td>

<td><input type="reset" name="submit" value="Reset" /></td>

</tr>

</table>

</form>

</body>

</html>
```

This code displays a simple form with ne text box. If user do not enter the value and press the submit button then the form gives and error. The output is as follows:

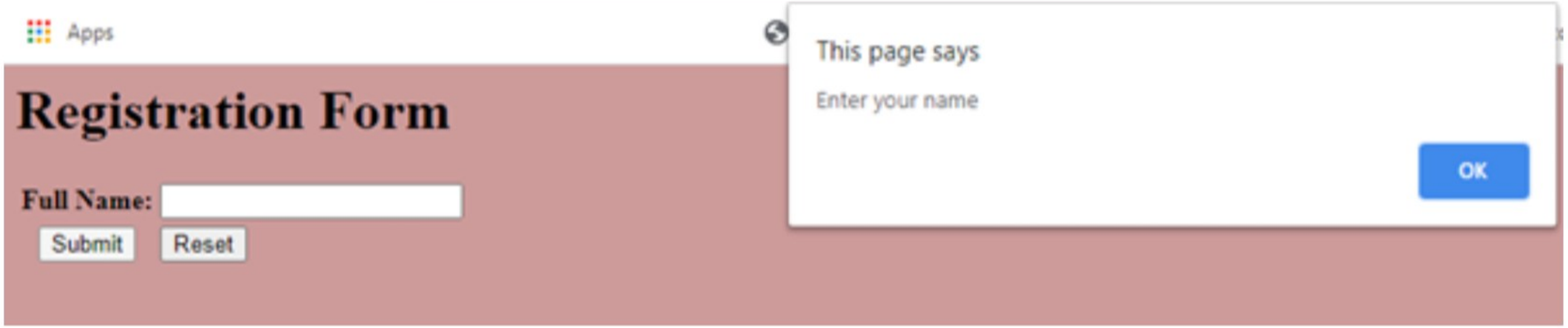


Figure 3: Output of the code in Example 9.8

Figure 3: Output of the code in Example 9.8

Some advantages and disadvantages of scripting languages are as follows:

Advantages:

- It easy to learn and use.
- It requires minimum programming knowledge.
- It allows complex tasks to be performed in relatively few steps
- It allows simple creation and editing in a variety of text editors
- It allows the addition of dynamic and interactive activities to web pages editing and running code is fast.

Disadvantages:

- The executable code can be downloaded from a server to a web browser's machine.
- To run code is installed and run by using the local browser's interpreter.
- By visiting doubtful web sites, the code can be downloaded without validating authenticity.
- The formal rules define scripting languages like JavaScript and VBScript.

9.5 Innovative Features of Scripting Languages

Most scripting languages do not require variables to be declared like Scheme. Unlike other languages such as Perl and JavaScript allow optional declarations, sort of compiler and checked the documentation. Some scripting languages use dynamic typing and by using strict 'vars' Perl can be run in a mode which may require declarations. At the run time the interpreter can perform type checking. There are some innovative features of scripting languages are as follows:

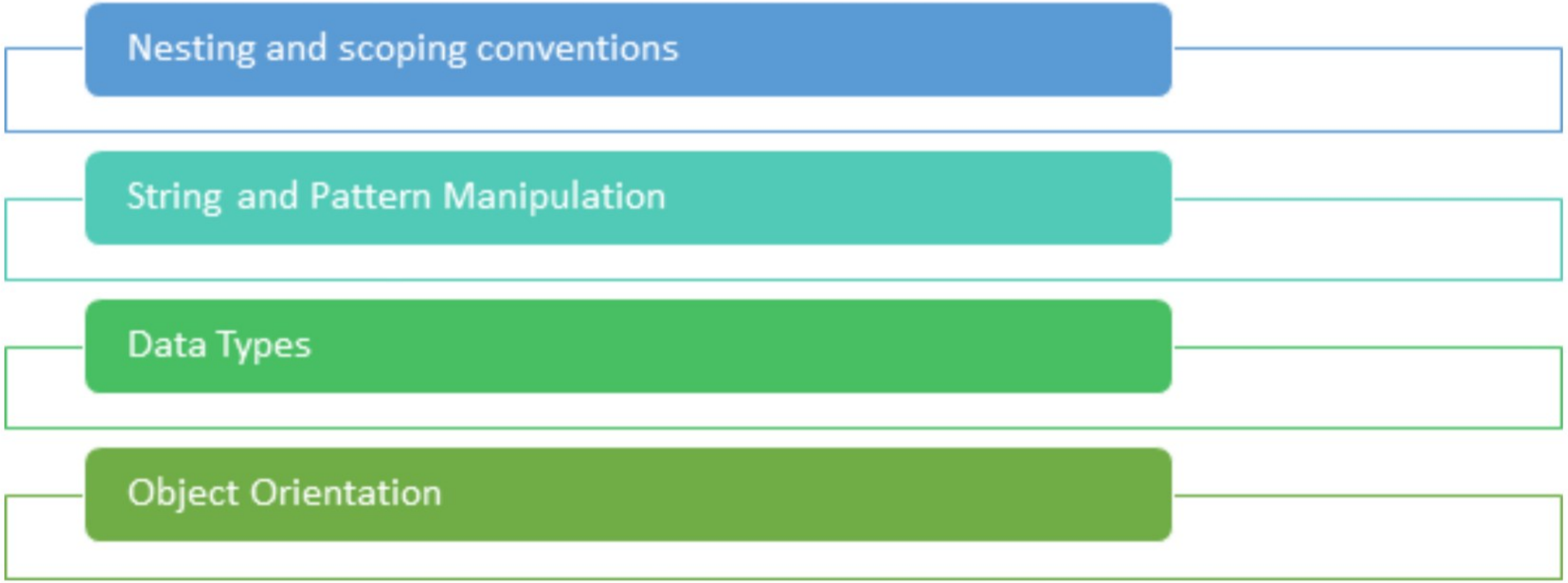


Figure 3: Innovative features

- **Nesting and scoping conventions:** These vary quite a bit like Scheme, Python, JavaScript languages provide the combination of nested subroutines and static scoping and Tcl provide subroutines to nest and uses dynamic scoping.

Figure 3: Innovative features

- **Nesting and scoping conventions:** These vary quite a bit like Scheme, Python, JavaScript languages provide the combination of nested subroutines and static scoping and Tcl provide subroutines to nest and uses dynamic scoping.
- **String and Pattern Manipulation:** In many scripting languages regular expressions are present and related tools provide extended versions of the notation in sed, awk, Perl, Tcl, Python, and Ruby grep. There are two main groups are:
    - In the first group, it includes awk, egrep and regex routines for C standard library. In the POSIX standard REs are implemented.
    - In the second group, Perl provides a large set of extensions and also referred to as an advanced REs.
- **Data types:** In most scripting languages they do not require the declaration of types for variables. For example, Scheme, Python, and Ruby languages perform extensive run-time checks to avoid repeated values. Where, Perl, Rexx and Tcl accept programmers' position which checks for the errors.
- **Object Orientation:** An object-oriented style has features in which it allows the programmer to use a more traditional style. For example, Perl 5, PHP and JavaScript have features which allow the programmer to clean programs. Where object-oriented conventional features allow both the Python and ruby programmer to be uniformly object-oriented.

9.6 Recap and Questions

9.6.1 Recap

Let us quickly recapitulate the following discussion points:

- A scripting language makes for specific runtime environments where it automates some of the code which does not require the compilation step.
- Server-side scripting languages run on a web server.
- Client-side scripting languages run on the client end on their web browser.
- Problem domain refers to all information which defines the problem and solution.
- Scripting languages are becoming more popular due to the emergence of web-based applications.
- At the run time the interpreter can perform type checking.

9.6.2 Test Your Skills- Multiple choice questions

1. Which language translates code into the machine?

a. JavaScript
b. Scripting
c. Python
d. Ruby

9.6.2 Test Your Skills- Multiple choice questions

1. Which language translates code into the machine?

a. JavaScript
b. Scripting
c. Python
d. Ruby

2. Name the common characteristics of scripting language focuses on reading source program before producing an output.

a. Cost-effective
b. High data types
c. Interactive
d. Dynamic typing

3. Name the one regular expression which includes a routine for C standard library.

a. Awk
b. Rexx
c. Tcl
d. Perl

4. Which programming language performs uniformly object-oriented?

a. Perl

4. Which programming language performs uniformly object-oriented?

a. Perl
b. PHP
c. JavaScript
d. Python

5. Which option reduces the demand on the server by allowing web pages to load faster?

a. Client
b. Server
c. Programmer
d. Language

9.6.3 Fill in the blanks

1. A client sends a _____ to the server and it responds via HTTP.
2. _____ programming language accept all commands from the keyboard
3. _____describe rules for essential functions of a solution product.
4. A scripting language prefers _____ level language.
5. Perl and python are known as _____ language.

9.6.4 Theoretical Questions

c. Programmer
d. Language

9.6.3 Fill in the blanks

1. A client sends a _____ to the server and it responds via HTTP.
2. _____ programming language accept all commands from the keyboard
3. _____describe rules for essential functions of a solution product.
4. A scripting language prefers _____ level language.
5. Perl and python are known as _____ language.

9.6.4 Theoretical Questions

1. Define the term scripting language.
2. Explain in detail the common characteristics of scripting language.
3. What are the different problems domains of a scripting language?
4. Describe the use of scripting in web development.
5. Discuss the innovative features of scripting languages.