Question Bank Answers (Paradigms & Computer Programming Fundamentals - IAE2)

> *CONTENT WARNING:*
> *READING THIS DOCUMENT MAY CAUSE SUDDEN BURSTS OF INTELLIGENCE.*
> *PROCEED WITH CAUTION.*

1. What are scripting languages? Explain characteristics of scripting languages.

Scripting Languages:

Scripting languages are programming languages designed for integrating and communicating with other programming languages, typically within a specific environment. They are often used for automating tasks, controlling software applications, and processing data. Scripting languages are high-level programming languages that are interpreted rather than compiled.

Characteristics of Scripting Languages:
i. Ease of Use:

They typically have simpler syntax compared to compiled languages, making them more accessible for beginners and allowing for quicker development.

ii. Interpreted:
Scripting languages are generally interpreted rather than compiled, which means the code is executed line-by-line at runtime, allowing for more flexibility and ease of debugging.

iii. Rapid Development:
Scripting languages support rapid prototyping and development due to their high-level nature and extensive libraries, enabling developers to write less code to accomplish tasks.

iv. Dynamic Typing:
Most scripting languages use dynamic typing, meaning variable types are determined at runtime, allowing for more flexibility in coding but potentially leading to runtime errors.

v. Platform Independence:
Many scripting languages can run on multiple platforms without modification, enhancing portability and usability across different systems.

Examples of Scripting Languages:

- **JavaScript**: Primarily used for web development to create interactive effects within web browsers.
- **Python**: Known for its readability and broad applicability, from web development to data analysis.

2. Explain the different communication and synchronization techniques in concurrent programming model.

In concurrent programming, managing communication and synchronization between processes or threads is crucial to ensure data integrity and proper execution.

Communication Techniques:

1. **Message Passing**:
   - Processes communicate by sending and receiving messages. This can be done using:
     - **Synchronous Message Passing**: The sender waits for the receiver to acknowledge receipt before continuing.
     - **Asynchronous Message Passing**: The sender continues executing without waiting for the receiver.
2. **Shared Memory**:
   - Multiple processes access a common memory space to communicate. This requires careful management to prevent conflicts.
3. **Sockets**:
   - Used for communication over a network, allowing processes on different machines to exchange data.

Synchronization Techniques:
Blocking synchronization
is a technique in which a thread or process is forced to stop (or "block") its execution when trying to access a resource that is currently unavailable or being used by another thread. The blocked thread waits until the resource becomes available again, and only then resumes its execution.

Key Characteristics:

The thread **suspends** its execution while waiting.
The thread remains in a
waiting state until the resource is available.
A
lock, **semaphore**, or **condition variable** is often used to implement blocking synchronization.

1. **Mutexes (Mutual Exclusion Locks)**:

- A lock that allows only one thread to access a resource at a time. It prevents race conditions by ensuring that critical sections are executed by only one thread.

2. **Semaphores**:
   - A signaling mechanism that can control access to a resource. They can be:
     - **Binary Semaphores**: Similar to mutexes, allowing only one thread to access a resource.
     - **Counting Semaphores**: Allow multiple threads to access a limited number of resources.
3. **Monitors**:
   - A higher-level abstraction that combines mutual exclusion and the ability to wait (suspend) for conditions to be true. Monitors encapsulate shared data and provide methods for safe access.

3. Explain the need for thread synchronization in concurrent programming.

Thread synchronization is crucial in concurrent programming to ensure that multiple threads operate safely and efficiently when sharing resources or interacting with each other.

Thread synchronization is essential in concurrent programming for several reasons:

1. **Preventing Race Conditions**

Race Conditions occur when two or more threads access shared data simultaneously and at least one thread modifies it. Without synchronization, the final state of the data can become unpredictable and may lead to incorrect results or system crashes.

2. **Maintaining Data Integrity**

When multiple threads modify shared resources (like variables, data structures, or files), synchronization ensures that data remains consistent and valid. This is crucial in applications where accuracy is critical, such as financial systems or databases.

3. **Coordinating Execution Order**

In certain scenarios, the execution order of threads matters. Synchronization techniques (like barriers and condition variables) allow threads to wait for specific conditions or events, ensuring that tasks are completed in the correct order.

4. **Avoiding Deadlocks**

Deadlocks occur when two or more threads are waiting indefinitely for each other to release resources. Proper synchronization techniques can help design systems that avoid these situations by managing resource allocation effectively.

5. **Resource Management**

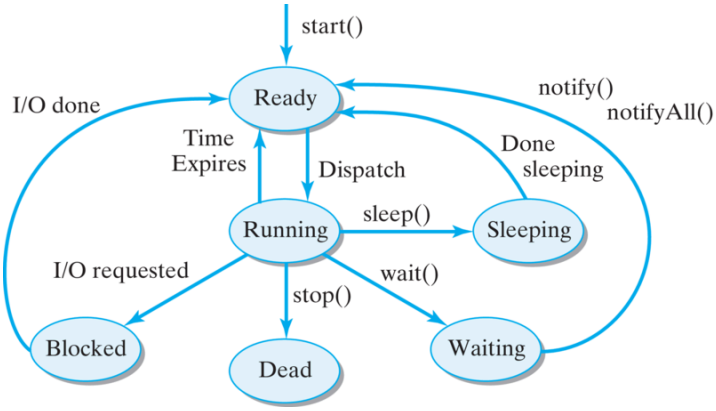
Threads may need access to limited resources (like database connections or file handles). Synchronization helps manage access to these resources, ensuring that only a designated number of threads can use them simultaneously.

7. **Improving Performance**

While it may seem counterintuitive, effective synchronization can improve performance by reducing the overhead of managing shared resources. It allows threads to work concurrently without stepping on each other's toes, leading to faster overall execution.

4. Explain the life cycle of a thread with neat labeled diagram.

The life cycle of a thread typically consists of several states that a thread can be in during its existence.



Thread Life Cycle States

1. **New (or Created):**
   - A thread is in the New state when it is created but not yet started. At this point, resources are allocated, but the thread has not begun execution.
2. **Runnable:**
   - Once the thread is started (using methods like `start()` in Java), it enters the Runnable state. In this state, the thread is ready to run and may be actively executing or waiting for CPU time to be allocated.
3. **Blocked:**
   - A thread enters the Blocked state when it is waiting for a monitor lock to enter a synchronized block or method. It cannot proceed until the lock is released by another thread.
4. **Waiting:**
   - A thread is in the Waiting state when it is waiting for another thread to perform a specific action, such as sending a notification (using methods like `wait()` or `join()`). It will remain in this state until it receives the necessary signal.
5. **Timed Waiting:**
   - Similar to the Waiting state, a thread in Timed Waiting is waiting for a specified period. This can occur when using methods like `sleep(milliseconds)` or `wait(milliseconds)`. If the time expires, the thread will return to the Runnable state.
6. **Terminated (or Dead):**
   - A thread enters the Terminated state when it has completed its execution, either because it finished normally or was terminated due to an error or an exception.

5. List various characteristics of scripting languages?

Characteristics of Scripting Languages:

i. Ease of Use

:

They typically have simpler syntax compared to compiled languages, making them more accessible for beginners and allowing for quicker development.

ii. Interpreted:

Scripting languages are generally interpreted rather than compiled, which means the code is executed line-by-line at runtime, allowing for more flexibility and ease of debugging.

iii. Rapid Development:

Scripting languages support rapid prototyping and development due to their high-level nature and extensive libraries, enabling developers to write less code to accomplish tasks.

iv. Dynamic Typing:
Most scripting languages use dynamic typing, meaning variable types are determined at runtime, allowing for more flexibility in coding but potentially leading to runtime errors.

v. Platform Independence:
Many scripting languages can run on multiple platforms without modification, enhancing portability and usability across different systems.

Examples of Scripting Languages:

- **JavaScript**: Primarily used for web development to create interactive effects within web browsers.
- **Python**: Known for its readability and broad applicability, from web development to data analysis.

6. Write difference between process and threads

Process	Thread
Process means any program is in execution.	Thread means a segment of a process.
The process takes more time to terminate.	The thread takes less time to terminate.
It takes more time for creation.	It takes less time for creation.
It also takes more time for context switching.	It takes less time for context switching.
The process is less efficient in terms of communication.	Thread is more efficient in terms of communication.
The process is isolated.	Threads share memory.
The process is called the heavyweight process.	A Thread is lightweight as each thread in a process shares code, data, and resources.
Process switching uses an interface in an operating system.	Thread switching does not require calling an operating system and causes an interrupt to the kernel.
A system call is involved in it.	No system call is involved, it is created using APIs.
The process does not share data with each other.	Threads share data with each other.

7. What do you mean by deadlock? write the necessary condition for deadlock to occur.

What is Deadlock?

A **deadlock** is a situation in concurrent programming where two or more threads (or processes) are unable to proceed because each is waiting for the other to release a resource. This results in a standstill where none of the involved threads can continue execution, leading to resource wastage and application stalls.

Necessary Conditions for Deadlock

For a deadlock to occur, the following four necessary conditions must all be true simultaneously:

Mutual Exclusion:

- At least one resource must be held in a non-shareable mode. If another thread requests that resource, it must be blocked until the resource is released by the holding thread.

Hold and Wait:

- A thread is holding at least one resource and is waiting to acquire additional resources that are currently being held by other threads. This condition implies that threads can hold resources while waiting for others.

No Preemption:

- Resources cannot be forcibly taken from a thread holding them. A resource can only be released voluntarily by the thread holding it after it has completed its task.

Circular Wait:

- There exists a set of threads {T1,T2,...,Tn} such that:
  - T1 is waiting for a resource held by T2,
  - T2 is waiting for a resource held by T3,
  - ...
  - Tn is waiting for a resource held by T1.
- This creates a circular chain of dependencies that leads to deadlock.

8. Explain the term :race condition, critical section

Race Condition:

A **race condition** occurs in a concurrent programming environment when two or more threads or processes attempt to access shared resources simultaneously, and at least one of them modifies the resource. The final outcome depends on the sequence or timing of the threads' execution, leading to unpredictable behavior and potentially incorrect results.

Example of Race Condition:

Consider a scenario where two threads increment the same variable:

```
# Shared variable
counter = 0

# Thread 1
counter += 1

# Thread 2
counter += 1
```

If both threads read the variable at the same time, they might both read `0`, increment it to `1`, and then write `1` back, resulting in a final value of `1` instead of `2`. This unpredictable behavior is a race condition.

Critical Section

A **critical section** is a segment of code that accesses shared resources and must not be executed by more than one thread or process at the same time. It is a part of the program where shared data is accessed, and ensuring that only one thread can enter this section at any given time is crucial to prevent race conditions.

Characteristics of Critical Sections:

Mutual Exclusion: Only one thread can execute in the critical section at any given time.

Progress: If no thread is executing in the critical section and there are threads that wish to enter, then only those threads that are not blocked can make progress.

Bounded Waiting: There exists a limit on the number of times other threads can enter their critical sections after a thread has requested to enter its critical section and before that request is granted.

Example: If multiple threads are updating the balance of a shared bank account, the section of code that modifies the balance should be a critical section, so only one thread can access and modify it at a time.

Solution: Synchronization mechanisms like locks, semaphores, or mutexes are used to ensure that only one thread enters the critical section at a time, thereby preventing race conditions and ensuring data integrity.

9. What are different ways to create threads?

Creating threads can be accomplished in various ways, depending on the programming language and its threading model.

Extending the Thread Class:

In languages like Java, you can create a thread by defining a new class that extends the `Thread` class and overriding its `run()` method.

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running");
    }
}
MyThread thread = new MyThread();
thread.start();
```

Implementing the Runnable Interface

Another way to create a thread in Java is to implement the `Runnable` interface. This allows your class to be more flexible, as it can extend another class while still being runnable in a thread.

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is running");
    }
}
Thread thread = new Thread(new MyRunnable());
thread.start();
```

Other Methods:

- **Using OS APIs (C/C++)**: Create threads using POSIX threads.

  ```
  pthread_create(&thread, NULL, thread_function, NULL);
  ```

- **Using the Threading Module (Python)**: Create threads using `threading.Thread`.

  ```
  thread = threading.Thread(target=thread_function)
  thread.start()
  ```

10. Explain background and motivation of concurrent programming

Background of Concurrent Programming:
Concurrent programming arose from the need to improve the efficiency and performance of software applications, especially as computing hardware evolved. Here are some key points in its development:

1. **Historical Context**:
   - Early computing was largely sequential, with one process executing at a time. As hardware capabilities grew, particularly with the advent of multi-core and multi-processor systems, the need for software to exploit this parallelism became apparent.
2. **Emergence of Multi-Core Processors**:
   - The transition from single-core to multi-core processors allowed multiple threads or processes to execute simultaneously. This shift necessitated programming models that could manage concurrency effectively to

take full advantage of available resources.

3. **Complex Systems**:
    - Modern applications, such as web servers, database systems, and real-time systems, often require handling multiple tasks simultaneously (e.g., processing user requests, managing I/O operations). Concurrent programming enables these applications to remain responsive and efficient.

Motivation for Concurrent Programming

The primary motivations for concurrent programming include:

Performance Improvement:

- By executing multiple tasks in parallel, applications can significantly reduce execution time, especially for CPU-bound and I/O-bound tasks.

Resource Utilization:

- Concurrent programming allows for better utilization of system resources, maximizing the capabilities of multi-core processors and distributed systems.

Responsiveness:

- In interactive applications (like GUIs), concurrent programming helps maintain responsiveness. While one thread handles user input, another can perform background tasks without freezing the interface.

Scalability:

- Applications designed with concurrency in mind can easily scale to handle increased workloads. Adding more threads or processes can improve performance without significant redesign.

Real-Time Processing:

- Many applications require real-time processing capabilities (e.g., embedded systems, robotics). Concurrent programming allows for meeting strict timing requirements by managing tasks effectively.

Handling I/O Operations:

- Concurrent programming is particularly useful for applications that perform extensive I/O operations, allowing the program to continue executing while waiting for I/O to complete.

11. Give examples of popular scripting languages.

Examples of popular scripting languages include:

1. **Python:** Known for its simplicity and readability, Python is widely used in fields such as web development, data analysis, scientific computing, and automation.

2. **JavaScript**: Primarily used for web development, JavaScript allows for client-side scripting in web browsers and server-side scripting with Node.js.

3. **Ruby**: Ruby is known for its elegant syntax and is often used in web development, particularly with the Ruby on Rails framework.

4. **Perl:** Perl is a versatile scripting language used in system administration, text processing, and web development.

5. **Bash**: The Bourne-Again Shell (Bash) is a scripting language commonly used for system administration tasks and automation on Unix-like operating systems.

6. **PowerShell**: Developed by Microsoft, PowerShell is a scripting language designed for system administration and automation on Windows.

12. What are different problem domain for using scripting

Scripting languages are versatile and can be applied across various problem domains. Here are some key areas where scripting is commonly used:

1. **Web Development**

Client-Side: JavaScript is used to create interactive web pages, enhancing user experience with dynamic content.

Server-Side: Languages like PHP and Python (with frameworks like Flask or Django) are used to handle backend operations, manage databases, and serve web applications.

2. **Automation and Scripting Tasks**

System Administration: Shell scripts (e.g., Bash) are used to automate routine tasks such as backups, user management, and software installation on Unix/Linux systems.

Task Automation: Python and PowerShell are often used for automating repetitive tasks in various environments, including file manipulation and report generation.

3. **Data Analysis and Processing**

Data Manipulation: Python (with libraries like Pandas and NumPy) and R are extensively used for data cleaning, analysis, and visualization in data science.

Scripting for Big Data: Languages like Python and R are used in data processing frameworks (e.g., Apache Spark) to handle large datasets.

4. **Game Development**

Game Scripting: Languages like Lua and Python are used for scripting game logic and behavior, allowing developers to easily modify game functionality without recompiling.

6. **Web Scraping**

Data Extraction: Python, with libraries like Beautiful Soup and Scrapy, is often used for web scraping to collect data from websites for analysis or reporting.

7. **Machine Learning and Artificial Intelligence**

Prototyping Models: Python is the dominant language in machine learning, providing libraries like TensorFlow and Scikit-learn for building and training models quickly.

13. Explain use of scripting in web development

Scripting plays a crucial role in web development, enhancing both client-side and server-side functionality. Here's an overview of how scripting is used in different aspects of web development:

1. **Client-Side Scripting**

Client-side scripting involves code that runs in the user's browser. It primarily enhances user interaction and experience.

JavaScript:

- The most common client-side scripting language, JavaScript is used to create dynamic and interactive web pages.
- **Examples**:
  - **Form Validation**: Ensures that user input is correct before submitting forms, improving user experience.
  - **Dynamic Content Updates**: Enables content changes without reloading the page (e.g., loading more articles on scroll).

2. **Server-Side Scripting**

Server-side scripting involves code that runs on the web server, handling backend operations and data processing.

PHP:

- A widely used server-side scripting language for creating dynamic web pages. PHP can interact with databases and manage sessions.

- **Examples**:
  - **User Authentication**: Processes user login and registration.
  - **Database Interaction**: Fetches and displays data from a database (e.g., MySQL).

Python:

- Often used with frameworks like Flask and Django, Python simplifies web development by providing powerful tools for routing, templates, and database management.
- **Examples**:
  - **REST APIs**: Building APIs for client applications to interact with server data.

3. **Database Interaction**

Scripting languages facilitate communication between web applications and databases.

SQL:

- While not a scripting language in the traditional sense, SQL queries are often embedded within server-side scripts (like PHP or Python) to interact with relational databases.

5. **Content Management Systems (CMS)**

Scripting languages are foundational for CMS platforms like WordPress, Joomla, and Drupal.

- **Dynamic Content Generation**:
  - PHP and other server-side languages dynamically generate HTML content based on user interactions and database queries.

6. **Security and Session Management**

Scripting languages help manage user sessions and security features.

Session Handling:

- Languages like PHP provide built-in support for managing user sessions, ensuring secure access to user data.

Input Sanitization:

- Scripting is used to validate and sanitize user input to prevent security vulnerabilities, such as SQL injection and cross-site scripting (XSS).

14. Explain client side scripting and list the scripting languages used for it.

Client-Side Scripting

Client-side scripting refers to code that is executed in the user's web browser rather than on the web server. This type of scripting enhances user interaction and provides dynamic content without the need for continuous communication with the server. It allows for a responsive user experience and reduces server load by offloading certain tasks to the client.

Key Features of Client-Side Scripting

Interactivity: Client-side scripts enable interactive features, such as form validation, animations, and dynamic content updates, improving the overall user experience.

Reduced Server Load: By handling user interactions directly in the browser, client-side scripts can reduce the number of requests sent to the server, decreasing server load and response times.

Immediate Feedback: Users receive immediate feedback without waiting for a round trip to the server, making applications feel more responsive.

Asynchronous Processing: Using techniques like AJAX (Asynchronous JavaScript and XML), client-side scripts can communicate with the server in the background, allowing for partial page updates.

- **Form Validation**: Checking user input before submission to ensure it meets certain criteria (e.g., valid email format).
- **Dynamic Content**: Loading new content (e.g., articles, images) without refreshing the entire page.
- **User Interface Enhancements**: Creating effects like dropdown menus, modal windows, and animations.

Scripting Languages Used for Client-Side Scripting

JavaScript:

- The primary and most widely used client-side scripting language, enabling dynamic content and interactivity on web pages.
- Supported by all major web browsers.

TypeScript:

- A superset of JavaScript that adds static typing and other features, making it easier to manage larger codebases. Compiles down to JavaScript for browser compatibility.

VBScript:

- Primarily used in Microsoft environments (like Internet Explorer) for client-side scripting. However, its use has significantly declined with the rise of more modern alternatives.

HTML (with embedded scripting):

- While HTML itself is not a scripting language, it often contains embedded scripts (usually JavaScript) that enhance the interactivity of web pages.

CSS (Cascading Style Sheets):

- While not a scripting language per se, CSS can be manipulated through client-side scripts (like JavaScript) to create dynamic styling effects.

15. Explain server side scripting and list the scripting languages used for it

Server-Side Scripting:
Server-side scripting
refers to code that is executed on the web server before the content is sent to the user's browser. This approach allows for dynamic content generation, database interaction, and business logic processing, providing a tailored experience for each user. Server-side scripts process requests, manage sessions, and handle data storage and retrieval.

Key Features of Server-Side Scripting

Dynamic Content Generation: Server-side scripts can create HTML, JSON, or other formats dynamically based on user inputs or database queries.

Database Interaction: Server-side scripting allows direct communication with databases, enabling CRUD (Create, Read, Update, Delete) operations to manage data.

User Authentication: It handles user login and session management, ensuring secure access to resources.

Business Logic: Server-side scripts can implement complex business rules and algorithms that are executed before delivering the response to the client.

Data Processing: They can perform calculations, generate reports, and manipulate data before sending it to the client.

Common Use Cases:

Web Application Development: Creating applications that require user interactions, such as e-commerce sites, social networks, and content management systems.

API Development: Building RESTful APIs that serve data to client applications, mobile apps, or other services.

Scripting Languages Used for Server-Side Scripting

PHP:

- A widely used server-side scripting language that excels in web development. It integrates easily with databases like MySQL and is commonly used in content management systems (e.g., WordPress).

Python:

- Often used with frameworks like Django and Flask, Python provides powerful tools for web development and data handling. It is known for its readability and versatility.

Ruby:

- Typically used with the Ruby on Rails framework, Ruby simplifies web application development by emphasizing convention over configuration.

Java:

- While traditionally associated with enterprise applications, Java can be used for server-side scripting through frameworks like Spring and JavaServer Pages (JSP).

Node.js:

- A JavaScript runtime that allows for server-side scripting using JavaScript. It is particularly well-suited for building scalable network applications.

16. Explain innovative features of scripting languages

Scripting languages have evolved significantly over the years, incorporating various innovative features that enhance their usability, flexibility, and performance. Here are some of the key innovative features commonly found in modern scripting languages:

1. **Interpreted Execution**

Description: Scripting languages are typically interpreted rather than compiled, allowing for immediate execution of code without a separate compilation step.

Benefit: This enables rapid development and testing, as developers can quickly iterate and see results without the delay of compilation.

2. **Dynamic Typing**

Description: Many scripting languages support dynamic typing, allowing variables to hold data of any type without explicit declaration.

Benefit: This flexibility speeds up development and reduces boilerplate code, though it may lead to runtime errors if not managed carefully.

3. **Rich Standard Libraries**

Description: Many scripting languages come with extensive standard libraries that provide built-in functions for various tasks, such as file handling, networking, and data manipulation.

Benefit: Developers can leverage these libraries to avoid reinventing the wheel, accelerating development time and enhancing code quality.

4. **Support for Multiple Paradigms**

Description: Scripting languages often support multiple programming paradigms, including procedural, object-oriented, and functional programming.

Benefit: This versatility allows developers to choose the most appropriate paradigm for their specific task, leading to more effective and maintainable code.

5. **Concurrency Support**

Description: Modern scripting languages often provide built-in support for concurrency and parallelism, enabling the execution of multiple threads or asynchronous tasks.

Benefit: This is essential for building responsive applications, especially in web and network programming.

6. **Cross-Platform Compatibility**

Description: Many scripting languages are designed to run on various platforms without modification (e.g., Windows, macOS, Linux).

Benefit: This enhances portability, allowing developers to write code once and run it anywhere.

7. **Lightweight Syntax**

- **Description**: Scripting languages often feature a more concise and readable syntax compared to traditional programming languages.
- **Benefit**: This makes the code easier to write and maintain, especially for beginners and rapid development.

17. Function in Haskell

Haskell Functions

Functions play a major role in Haskell, as it is a functional programming language. Like other languages, Haskell does have its own functional definition and declaration.
Function declaration consists of the function name and its argument list along with its output.
Function definition is where you actually define a function.

```
add :: Int -> Int -> Int
add x y = x + y
putStrLn "The addition of the two numbers is:"
print(add 2 5) --calling a function
```

18. Higher order function examples

- It's a function that takes another function as an argument.
- even is a first-order function.
- map and filter are higher-order functions.
- The functions which take at least one function as parameter or returns a function as it results or performs both is called Higher Order Function.
- Many languages including- Javascript, Go, Haskell, Python, C++, C# etc, supports Higher Order Function.
- It is a great tool when it comes to functional programming.

Example:

```
Prelude> even 1
False
Prelude> even 2
True
Prelude> map even [1,2,3,4,5]
[False,True,False,True,False]
Prelude> filter even [1,2,3,4,5]
[2,4]
```

Advantages of Higher Order Functions:

- By use of higher order function, we can solve many problems easily. For example we need to build a function which takes a list and another function as it's input, applies that function to each element of that list and returns the new list.
- In Haskell, this could be done really easily using the inbuilt higher order function called map.

19. Monoid

- Haskell defines everything in the form of functions.
- In functions, we have options to get our input as an output of the function.
- This is what a Monoid is.
- A Monoid is a set of functions and operators where the output is independent of its input.
- Let's take a function (*) and an integer (1).
- Now, whatever may be the input, its output will remain the same number only.
- That is, if you multiply a number by 1, you will get the same number.

In Haskell, a monoid is a type with a rule for how two elements of that type can be combined to make another element of the same type.

To be a monoid there also needs to be an element that you can think of as representing 'nothing' in the sense that when it's combined with other elements it leaves the other element unchanged.

A great example is lists.

Given two lists, say [1,2] and [3,4], you can join them together using ++ to get [1,2,3,4].

There's also the empty list [].

Using ++ to combine [] with any list gives you back the same list, for example []++[1,2,3,4]==[1,2,3,4].

Monoid Example

```
multi:: Int->Int
multi x = x * 1
add :: Int->Int
add x = x + 0
main = do
print(multi 9)
print (add 7)
```

20. Modules

If you have worked on Java, then you would know how all the classes are bound into a folder called package.

Similarly, Haskell can be considered as a collection of modules.

Haskell is a functional language and everything is denoted as an expression, hence a Module can be called as a collection of similar or related types of functions.

You can import a function from one module into another module.

All the "import" statements should come first before you start defining other functions

List Module

List provides some wonderful functions to work with list type data. Once you import the List module, you have a wide range of functions at your disposal.

In the following example, we have used some important functions available under the List module.

```
import Data.List
main = do
putStrLn("Different methods of List Module")
print(intersperse '.' "Tutorialspoint.com")
print(intercalate " " ["Lets","Start","with","Haskell"])
print(splitAt 7 "HaskellTutorial")
print (sort [8,5,3,2,1,6,4,2])
```

Here, we have many functions without even defining them.

That is because these functions are available in the List module.

After importing the List module, the Haskell compiler made all these functions available in the global namespace. Hence, we could use these functions.

Output:

Different methods of List Module
"T.u.t.o.r.i.a.l.s.p.o.i.n.t...c.o.m"
"Lets Start with Haskell"
("Haskell","Tutorial")
[1,2,2,3,4,5,6,8]

Char module

The Char module has plenty of predefined functions to work with the Character type. Take a look at the following code block –

```
import Data.Char
main = do
putStrLn("Different methods of Char Module")
print(toUpper 'a')
print(words "Let us study tonight")
print(toLower 'A')
```

Output:

Different methods of Char Module
'A'
["Let","us","study","tonight"]
'a'

21. List functions

Listing of functions:
Head Function
Tail Function
Last Function
Init Function
Null Function
Reverse Function
Length Function
Take Function
Drop Function
Maximum Function
Minimum Function

Sum Function
Product function
Elem function

1. **Head Function**

Returns the first element of a list.

```
head [3, 5, 8, 2, 1]  -- ⟹ 3
```

2. **Tail Function**

Returns all elements of a list except the first one.

```
tail [3, 5, 8, 2, 1]  -- ⟹ [5, 8, 2, 1]
```

3. **Last Function**

Returns the last element of a list.

```
last [3, 5, 8, 2, 1]  -- ⟹ 1
```

4. **Init Function**

Returns all elements of a list except the last one.

```
init [3, 5, 8, 2, 1]  -- ⟹ [3, 5, 8, 2]
```

5. **Null Function**

Checks if a list is empty, returning `True` or `False`.

```
null []               -- ⟹ True
null [3, 5, 8, 2, 1]  -- ⟹ False
```

6. **Reverse Function**

Reverses the order of elements in a list.

```
haskell
Copy code
reverse [3, 5, 8, 2, 1]  -- ⟹ [1, 2, 8, 5, 3]
```

7. **Length Function**

Returns the number of elements in a list.

```
haskell
Copy code
length [3, 5, 8, 2, 1]  -- ⟹ 5
```

8. **Maximum Function**

Returns the maximum element in a list.

```
maximum [3, 5, 8, 2, 1]  -- ⟹ 8
```

9. **Minimum Function**

Returns the minimum element in a list.

```
minimum [3, 5, 8, 2, 1]  -- ⟹ 1
```

10. **Sum Function**

Returns the sum of all elements in a list.

```
sum [1, 2, 3, 4, 5]  -- ⟹ 15
```

22. Files and streams

In Haskell, handling files and streams is essential for performing input and output (I/O) operations. Here's an overview of how files and streams work in Haskell, along with some examples.

Let us create a file and name it "abc.txt". Next, enter the following lines in this text file: "Welcome to Tutorialspoint. Here, you will get the best resource to learn Haskell."

Next, we will write the following code which will display the contents of this file on the console. Here, we are using the function readFile() which reads a file until it finds an EOF character.

```
main = do
let file = "abc.txt"
contents <- readFile file
putStrLn contents
```

The above piece of code will read the file "abc.txt" as a String until it encounters any End of File character.
This piece of code will generate the following output.
Welcome to Tutorialspoint
Here, you will get the best resource to learn Haskell.

23. First class functions

First class functions

Generally while programming, we use first class functions which means that the programming language treats functions as values – that you can assign a function into a variable, pass it around.

These functions do not take other functions as parameters and never has any function as their return type.

To overcome all these shortcomings of first-class functions, the concept of Higher Order function was introduced.

Characteristics of First-Class Functions

- **Can be Assigned to Variables**:
  Functions can be assigned to variables and used later.
- **Can be Passed as Arguments**:
  Functions can be passed as parameters to other functions.
- **Can be Returned from Other Functions**:
  Functions can return other functions.
- **Can be Stored in Data Structures**:
  Functions can be stored in lists or other data structures.

24. Features of haskell

1. **Purely Functional**

Haskell emphasizes pure functions, meaning functions always produce the same output for the same input and do not have side effects, leading to easier reasoning about code.

2. **Lazy Evaluation**

Haskell employs lazy evaluation, meaning expressions are not evaluated until their results are needed. This allows for the definition of infinite data structures and can improve performance by avoiding unnecessary calculations.

3. **Strong Static Typing**

Haskell has a strong, static type system that catches many errors at compile time. Type inference allows the compiler to deduce types automatically, making code both safe and concise.

4. **First-Class Functions**

Functions in Haskell are first-class citizens, allowing them to be passed as arguments, returned from other functions, and stored in data structures. This supports higher-order programming.

5. **Immutability**

In Haskell, data is immutable by default, meaning once a value is created, it cannot be changed. This reduces bugs related to state changes and makes reasoning about code easier.

6. **Pattern Matching**

Pattern matching is a powerful feature that simplifies the process of checking data structures against patterns, making code more readable and expressive.

7. **Concurrency and Parallelism**

Haskell provides strong support for concurrent and parallel programming, allowing developers to write programs that can take advantage of multicore processors easily.

8. **Type Classes**

Haskell's type class system enables polymorphism, allowing functions to operate on different types while maintaining type safety. This promotes code reuse and abstraction.

25. Pattern matching in haskell

Pattern Matching

Pattern Matching is process of matching specific type of expressions. It is nothing but a technique to simplify your code. This technique can be implemented into any type of Type class. If-Else can be used as an alternate option of pattern matching.

Pattern Matching can be considered as a variant of dynamic polymorphism where at runtime, different methods can be executed depending on their argument list. Live Demo

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact ( n - 1 )
main = do
putStrLn "The factorial of 5 is:"
print (fact 5)
```

The compiler will start searching for a function called "fact" with an argument. If the argument is not equal to 0, then the number will keep on calling the same function with 1 less than that of the actual argument.

When the pattern of the argument exactly matches with 0, it will call our pattern which is "fact 0 = 1". Our code will produce the following output –

The factorial of 5 is: 120

26. Lambda function examples

We sometimes have to write a function that is going to be used only once, throughout the entire lifespan of an application.

To deal with this kind of situations, Haskell developers use another anonymous block known as lambda expression or lambda function.

A function without having a definition is called a lambda function. A lambda function is denoted by "\" character.

Let us take the following example where we will increase the input value by 1 without creating any function.

```
main = do
putStrLn "The successor of 4 is:"
print ((\x -> x + 1) 4)
```

Here, we have created an anonymous function which does not have a name. It takes the integer 4 as an argument and prints the output value. We are basically operating one function without even declaring it properly. That's the beauty of lambda expressions.

Our lambda expression will produce the following output –
The successor of 4 is:
5

27. Higher order functions and list comprehension in haskell. Refer all examples in PPT

(Refer to and all other Haskell questions, because all the Haskell answers and examples are taken from ma'am's presentation)

Mohammed Anas Nathani