UTA ID: 1001982478
Shivani Panchiwala

CSE-5307-001-PROGRAMMING LANGUAGE
Homework 2

1) Why should you be interested in learning about Lambda Calculus?

Learning about lambda calculus can be valuable for a few reasons. Firstly, it provides a theoretical foundation for understanding functional programming languages like Haskell and Lisp, which are becoming increasingly popular in the industry. By understanding lambda calculus, you can gain a deeper understanding of how these languages work and how to use them effectively.

There are several reasons why one might be interested in learning about lambda calculus:

1. Understanding functional programming: Lambda calculus is the foundation of functional programming, a programming paradigm that emphasizes the use of functions to create programs. By learning lambda calculus, you can gain a deeper understanding of how functional programming languages work, such as Haskell or Lisp, and how to use them effectively.
2. Computational theory: Lambda calculus is a mathematical model of computation and is used to explore the theory of computation. By studying lambda calculus, you can gain insight into the nature of computation and the limitations of what computers can and cannot do.
3. Artificial intelligence and machine learning: Lambda calculus is used as a theoretical foundation for developing machine learning algorithms, such as deep learning models. By learning lambda calculus, you can understand the underlying principles of these advanced techniques and how they are developed.
4. Logic and reasoning: Lambda calculus can be used to formalize logical systems, such as first-order logic. By studying lambda calculus, you can improve your logical reasoning skills and develop a deeper understanding of formal logic.

Overall, learning about lambda calculus can be beneficial for anyone interested in computer science, programming, artificial intelligence, logic, or mathematics. It provides a foundation for understanding many advanced topics in these fields and can help you develop a deeper understanding of the underlying principles.

2) How do you encode the concepts of TRUE, FALSE, NOT, AND, OR?

In lambda calculus, we can encode the concepts of TRUE and FALSE using lambda expressions that represent Boolean values. Here are some examples:

TRUE: $\lambda x.\lambda y.x$ This lambda expression takes two arguments, x, and y, and returns x. This represents the concept of TRUE because no matter what the value of y is, the expression always returns x.

FALSE: λx.λy.y Similarly, this lambda expression takes two arguments, x, and y, and returns y. This represents the concept of FALSE because no matter what the value of x is, the expression always returns y.

We can encode the concepts of NOT, AND, and OR using logical operations on these Boolean values. Here are some examples:

NOT: λp.p FALSE TRUE This lambda expression takes a Boolean value p and returns the opposite value (i.e., NOT p). If p is TRUE, the expression returns FALSE, and if p is FALSE, the expression returns TRUE.

AND: λp.λq.p q FALSE This lambda expression takes two Boolean values p and q and returns their logical AND (i.e., p AND q). If both p and q are TRUE, the expression returns TRUE. Otherwise, it returns FALSE.

OR: λp.λq.p TRUE q This lambda expression takes two Boolean values p and q and returns their logical OR (i.e., p OR q). If either p or q is TRUE, the expression returns TRUE. Otherwise, it returns FALSE.

By encoding these logical operations in lambda calculus, we can use them to build more complex expressions and functions, such as conditional statements and loops.

3) What is important about the Lambda Calculus expression called 'Y Combinator'?

The Y combinator is an important concept in functional programming because it enables the creation of anonymous recursive functions. In functional programming, functions are treated as first-class citizens, meaning they can be passed around as arguments and returned as values. However, defining a recursive function anonymously, i.e., without a named identifier, is not straightforward in most functional programming languages.

The Y combinator solves this problem by providing a way to define anonymous recursive functions. It is a higher-order function that takes a function as an argument and returns a new function that is the recursive version of the original function. The recursive version can then be called with the same arguments as the original function, and it will repeatedly call itself until it reaches a base case.

The Y combinator is important because it allows functional programmers to write more concise and expressive code. It enables the creation of recursive functions without needing to define them explicitly, which can lead to cleaner and more modular code. Additionally, the Y combinator is a fundamental concept in lambda calculus, which is the mathematical foundation of functional programming.

Overall, the Y combinator is an essential concept in functional programming that enables the creation of anonymous recursive functions, leading to more concise and expressive code.

UTA ID: 1001982478
Shivani Panchiwala

4) Write the Y Combinator expression in Lambda Calculus.

The Y combinator expression in lambda calculus is as follows:

$Y = \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$

In this expression, **λ** represents the lambda symbol, which is used in lambda calculus to define anonymous functions. **f** represents the function that we want to make recursive. The expression **(λx.f (x x))** represents an anonymous function that takes an argument **x** and applies **f** to the expression **(x x)**. This expression **(x x)** represents the recursive call to the function **f**.

The Y combinator is defined as the application of two anonymous functions to **f**. The first anonymous function **(λx.f (x x))** is applied to itself recursively, creating an infinite loop of recursive calls. The second anonymous function **(λx.f (x x))** is the identity function, which returns its argument unchanged.

When **Y** is applied to a function **f**, it returns a new function that is the recursive version of **f**. The recursive version can then be called with the same arguments as the original function, and it will repeatedly call itself until it reaches a base case.

5) Where did the language 'Haskell' get its name?

The programming language Haskell is named after the logician Haskell Curry, who was a pioneer of the theory of functional programming. Curry's work on the foundations of mathematics and logic provided the basis for many of the concepts and techniques used in functional programming.

Haskell was designed in the late 1980s and early 1990s by researchers interested in creating a more expressive and modular programming language. They wanted to create a language that would allow programmers to write more concise and maintainable code by using functional programming concepts such as immutability and higher-order functions.

The design of Haskell was influenced by several other programming languages, including Lisp, ML (Machine Learning), and Miranda. The language was designed to be purely functional, meaning that all functions are side-effect free and have no mutable state. Haskell also includes support for lazy evaluation, which allows programmers to write more efficient and expressive code by only evaluating expressions when they are needed.

Overall, Haskell was designed to be a modern and expressive language that is well-suited for functional programming. Its name honors Haskell Curry, whose work on the foundations of logic and mathematics provided the basis for many of the concepts and techniques used in functional programming today.

6) In the video it was mentioned that Erlang was used to code what?

In the video, it was mentioned that the programming language Erlang was used to code the messaging infrastructure for the telecommunication company Ericsson's AXD301 switch. The AXD301 switch was a high-performance telecommunications switch that was used in the early 2000s to handle large volumes of data traffic for mobile networks.

Erlang was chosen for this project because of its support for concurrency and fault tolerance, which are important properties for building reliable and scalable telecommunications systems. Erlang's lightweight processes and message passing model allowed developers to build a highly concurrent and fault-tolerant messaging system that could handle large volumes of traffic with minimal downtime.

The success of the AXD301 switch and its messaging infrastructure helped to establish Erlang as a viable language for building scalable and reliable distributed systems. Today, Erlang is used in a wide range of industries, including telecommunications, finance, and gaming, where it is valued for its ability to handle large volumes of traffic and maintain important levels of uptime.

7) How is 'pattern matching' used?

Pattern matching is a technique used in programming to match patterns of data and perform different actions based on the structure of that data. It is commonly used in functional programming languages such as Haskell, but it is also used in other programming paradigms, including object-oriented programming.

In functional programming languages, pattern matching is used extensively to destructure data and extract its components. For example, in Haskell, a function that takes a list as input can use pattern matching to match on the empty list **[]** and non-empty list **(x:xs)**, where **x** is the head of the list and **xs** is the tail. This allows the function to perform different actions based on whether the list is empty or not.

Pattern matching can also be used to match more complex data structures, such as algebraic data types. In Haskell, algebraic data types are defined using the data keyword, and they can be pattern matched on to extract their components. For example, a Type, which represents a value that may or may not be present, can be pattern matched on to extract it is Just value or handle the Nothing case.

In object-oriented programming, pattern matching is often used with inheritance and polymorphism. For example, the switch statement in Java can be used to match the type of an object and perform different actions based on that type. This is often used in conjunction with inheritance and polymorphism to handle different cases in a modular and extensible way.

Overall, pattern matching is a powerful technique used in programming to match patterns of data and perform different actions based on that data. It is commonly used in functional programming languages to destructure data and extract its components, but it can also be used in other programming paradigms to handle different cases in a modular and extensible way.

8) Complete this sentence: "NP (Nondeterministic Polynomial) problems are hard to solve but easy to _____"

"NP problems are hard to solve but easy to verify." This is because NP (nondeterministic polynomial time) problems are problems that can be verified in polynomial time, but the best-known algorithms for solving them have exponential time complexity. In other words, given a solution to an NP problem, it is easy to verify whether the solution is correct, but finding a solution in the first place is much more difficult. This contrasts with P (polynomial time) problems, which can be both solved and verified in polynomial time. The question of whether P equals NP, or whether there are problems in NP that can be solved in polynomial time, is one of the most important open problems in computer science.

9) What is the example of an NP problem used in the video?

In the video "P vs NP on TV" by Computerphile, an example of an NP problem used is the subset sum problem. The subset sum problem is a classic problem in computer science and mathematics, which asks whether a given set of integers can be partitioned into two subsets such that the sum of the integers in each subset is equal. This problem is in the class of NP problems because given a proposed solution (i.e., a partition of the set into two subsets), it can be verified in polynomial time whether the solution is correct. However, the best-known algorithms for solving the subset sum problem have exponential time complexity, making it difficult to find solutions to large instances of the problem.

10) What are the TV shows mentioned in the video?

In the video "P vs NP on TV" by Computerphile, two TV shows are mentioned as examples to illustrate the concept of P vs NP. The first TV show is the game show "Let's Make a Deal," in which contestants are presented with a set of doors and must choose one of them. The host then opens one of the other doors, revealing a non-prize, and the contestant is given the option to switch their choice to the remaining door or stick with their original choice. The video uses this game show as an example to demonstrate how the probabilities of winning can be calculated using combinatorics.

The second TV show mentioned is "The Mentalist," a crime drama series in which the main character uses his powers of observation and deduction to solve crimes. The video uses this TV show as an example to illustrate the difference between P and NP problems, and how NP problems are difficult to solve because the number of possible solutions grows exponentially with the size of the problem.

11) Floating point numbers are essentially what?

Floating point numbers are a way to represent real numbers in a computer's memory. In the video "Floating Point Numbers" by Computerphile, it is explained that floating point numbers consist of two parts: a mantissa (also known as a significand), which represents the significant digits of the

number, and an exponent, which represents the scale or magnitude of the number. The mantissa and exponent are both stored as binary numbers in the computer's memory, and together they form a floating-point representation of the real number. Floating point numbers are used in many scientific and engineering applications where high precision is required, but they can also introduce errors due to rounding and other issues.

12) Computers perform scientific notation in what base?

Computers perform scientific notation in base 2 (binary). In the video "Floating Point Numbers" by Computerphile, it is explained that floating point numbers are represented in binary using scientific notation, with the mantissa and exponent stored separately in the computer's memory. The use of binary allows for efficient computation and storage of floating-point numbers, but it can also introduce errors due to rounding and precision issues, especially when dealing with decimal numbers that do not have exact binary representations.

13) What is the problem with adding 1/3 + 1/3 + 1/3 using base 10 and ignoring recurring numbers?

The problem with adding 1/3 + 1/3 + 1/3 using base 10 and ignoring recurring numbers is that the result will not be exact because 1/3 has a repeating decimal representation in base 10 (i.e., 0.33333...). In the video "Floating Point Numbers" by Computerphile, it is explained that computers represent floating point numbers in binary (base 2), and the same issue can occur when representing certain decimal numbers in binary. For example, the decimal number 0.1 has a repeating binary representation (i.e., 0.00011001100110011...) that cannot be stored exactly in a finite amount of memory. As a result, when performing arithmetic with floating point numbers, it is important to be aware of the potential for rounding errors and other issues that can arise due to the limitations of the representation.

14) What is 1/10 in base 2?

In base 2 (binary), the decimal number 1/10 cannot be represented exactly as a finite binary number, just as it has a repeating decimal representation in base 10. However, it can be approximated using a finite number of bits. In the video "Floating Point Numbers" by Computerphile, it is mentioned that the IEEE (Institute of Electrical and Electronics Engineers) 754 standard for floating point arithmetic uses a 64-bit representation for double precision floating point numbers, with 53 bits for the mantissa and 11 bits for the exponent. In this representation, the closest binary approximation to 1/10 is:

0.0001100110011001100110011001100110011001100110011001100110011010

This binary number has 54 bits, which is one more bit than the mantissa can hold, so it must be rounded to fit into the available bits. The rounding rule used by IEEE 754 is called "round to nearest, ties to even," which means that the number is rounded to the closest representable value, with ties being broken by choosing the one with an even least significant bit. In this case, the closest representable value is:

0.0001100110011001100110011001100110011001100110011001100110011010

which is the binary approximation to 0.1000000000000000055511151231257827021181583404541015625 in decimal.

15) What is the name of the function discussed in the video?

The name of the function discussed in the video "The Most Difficult Program to Compute?" by Computerphile is the Ackermann function. The Ackermann function is a mathematical function that takes two non-negative integer arguments and returns a non-negative integer. It is named after Wilhelm Ackermann, who studied it in the 1920s, and it is notable for being a simple recursive function that grows extremely quickly as its arguments increase. As a result, it is often used as a benchmark for testing the performance of computer programs and computing systems. In the video, the presenters discuss the difficulty of computing the Ackermann function, especially for large values of its arguments, and they demonstrate how it can quickly overwhelm even powerful computing resources.

16) Can Ackermann's function be coded using for or 'DO' loops?

Yes, the Ackermann function can be implemented using loops, such as for or "DO" loops, but the resulting code may be less efficient and more difficult to understand than a recursive implementation. In the video "The Most Difficult Program to Compute?" by Computerphile, the presenters demonstrate how to write a recursive implementation of the Ackermann function in several programming languages, including Python, Java, and Haskell. They note that recursive implementations can be more elegant and easier to understand, but that they may also be less efficient due to the overhead of function calls and stack management. However, they also mention that some programming languages, such as Haskell, can optimize tail-recursive functions like the Ackermann function so that they are as efficient as iterative implementations using loops. Overall, the choice of implementation depends on the specific requirements and constraints of the application and the programmer's preferences and expertise.

17) What is the value of Ackermann (4,1)?

The value of Ackermann (4,1) is 65533. In the video "The Most Difficult Program to Compute?" by Computerphile, the presenters demonstrate the difficulty of computing the Ackermann function, especially for large values of its arguments. They note that the value of Ackermann (4,1) is already quite large, and that computing higher values of the function quickly becomes infeasible even for powerful computers. They also mention that the Ackermann function is not only difficult to compute, but also has deep connections to other areas of mathematics and computer science, including computability theory and the complexity of algorithms.

18) How many minutes will the machine in the video take to calculate Ackermann (4,2)

In the video "The Most Difficult Program to Compute?" by Computerphile, the presenter notes that it would take the machine in the video approximately 30 minutes to calculate Ackermann (4,2). This is because the Ackermann function grows extremely quickly as its arguments increase, and even small increases in the values of the arguments can result in much larger values of the function. For example, the value of Ackermann (4,1) is already quite large (65533), and computing

Ackermann (4,2) requires many more recursive calls and much more computational resources. The presenter notes that the exponential growth of the Ackermann function makes it an interesting and challenging benchmark for testing the performance of computing systems.

19) The performance characteristic of Ackermann's function is described as what?

In the video "The Most Difficult Program to Compute?" by Computerphile, the presenters describe the performance characteristic of Ackermann's function as "recursive explosion" or "explosive recursion". This refers to the fact that as the function's arguments increase, the number of recursive calls made by the function grows very quickly, leading to exponential growth in the amount of memory and computation required to evaluate the function. In fact, the growth rate of the Ackermann function is so rapid that it quickly surpasses the capacity of any physical computing system to calculate it, even for small input values. This property of the Ackermann function makes it a challenging benchmark for evaluating the performance of computing systems and algorithms.

20) A loop nested in another loop has the performance characteristic of what?

In the video "Programming Loops vs Recursion" by Computerphile, the presenter notes that a loop nested inside another loop has the performance characteristic of "nested iteration", which means that the execution time of the loop grows proportionally to the product of the sizes of the two loops. In other words, if the outer loop has n iterations and the inner loop has m iterations, the total number of iterations will be n x m, and the execution time will grow proportionally to this value. This can lead to terribly slow execution times for nested loops with large numbers of iterations, especially if the body of the inner loop contains computationally intensive operations. The presenter notes that in some cases, recursion can be a more efficient and elegant solution than nested loops, because it allows the function to call itself with different arguments and avoid the need for nested iteration.

21) What was the limitation of Fortran mentioned in the video?

In the video "Programming Loops vs Recursion" by Computerphile, the presenter notes that one limitation of the Fortran programming language was its lack of support for recursion. Fortran was designed primarily for scientific and engineering applications, where loops and iterative algorithms were the dominant paradigm. As a result, the language did not include many of the advanced control structures found in later languages, such as conditionals and recursive function calls. This made it difficult or impossible to express certain algorithms and data structures in Fortran and led to the development of other languages like Lisp and C, which had more powerful control structures and better support for recursion.

22) What real-world use needs complex recursion?

In the video "Programming Loops vs Recursion" by Computerphile, the presenter notes that complex recursion can be useful in a variety of real-world applications, particularly in fields like computer science and mathematics. One example he gives is the parsing of natural language sentences, which can involve complex recursive algorithms to analyze the grammatical structure of the sentence and identify the relationships between different words and phrases. Another example is the evaluation of mathematical expressions, which can often be expressed recursively

in terms of simpler sub-expressions. In general, any problem that involves hierarchical or tree-like structures can be a suitable candidate for recursive solutions, because the recursive structure mirrors the underlying structure of the problem and can simplify the algorithmic design.

23) There was a need to have a language that could cope with what?

In the video "Why 'C' is so influential" by Computerphile, the presenter notes that the development of the C programming language was driven by a need for a language that could cope with the complexity of writing large-scale operating systems. At the time of its creation in the 1970s, most programming languages were designed for specific applications or problem domains, and lacked the low-level control and flexibility needed to implement complex system software. C was designed to bridge this gap, providing a low-level, systems-oriented language that was close enough to the hardware to allow efficient memory management and direct hardware access, but high-level enough to allow abstraction and modular programming. This made C a popular choice for developing operating systems, compilers, and other system software, and laid the foundation for many of the modern programming languages that followed.

24) C is most powerful when considered as the classical what?

In the video "Why 'C' is so influential" by Computerphile, the presenter notes that C is most powerful when considered as the classical "assembly language" for modern computer systems. This is because C provides low-level control over memory and hardware, allowing programmers to write code that is both efficient and portable across different hardware platforms. C is also highly expressive, with a small set of powerful language constructs that can be used to build complex systems. Finally, C has a rich ecosystem of libraries and tooling that make it a popular choice for building software in a variety of domains, from systems programming to web development. All these factors combine to make C a powerful and enduring language that has had a profound impact on the history of computing.

25) What are the names of the two fields of the 'THING' structure?

In the video "Essentials: Pointer Power! - Computerphile", the 'THING' structure contains two fields named 'value' and 'next'. 'value' is a pointer to the value stored in the current node of the linked list, while 'next' is a pointer to the next node in the list. The 'THING' structure is used to implement a linked list data structure using pointers in C.

26) What is the advantage of the 'Triple Ref Technique'?

In the video "Triple Ref Pointers - Computerphile", the advantage of the triple ref technique is that it allows for the modification of a pointer itself, rather than just the value it points to. This is useful in situations where a function needs to modify a pointer that has been passed to it as an argument. Without the triple ref technique, the function can only modify the value being pointed to, not the pointer itself. The triple ref technique allows the function to modify the pointer, ensuring that changes made to the pointer persist after the function returns.

27) What is the procedure used in the video to compare the different structures?

In the video "Arrays vs Linked Lists - Computerphile", the procedure used to compare the different structures is to measure the time it takes to perform certain operations on each structure. For example, to compare the performance of arrays and linked lists when searching for an element, a program is written to perform the search on both structures, and the time taken to complete the search is measured. The same procedure is used to compare the performance of the two structures when inserting or deleting elements. By comparing the times taken to perform these operations on each structure, it is possible to determine which structure is more efficient for a given task.

28) Why is the reverse array faster on the Atari?

In the video "Arrays vs Linked Lists - Computerphile", it is explained that the reverse array operation is faster on the Atari because the Atari had a hardware stack which could be utilized to traverse the array in reverse order, while linked lists required manual traversal of each element, which was computationally expensive on the Atari's hardware.

29) What would be the goal of requiring people to be exposed to coding?

The goal of requiring people to be exposed to coding would be to give them a basic understanding of how technology works and how to interact with it. It would also help them develop problem-solving skills, logical thinking, and creativity, which are useful not only in programming but also in other fields. Furthermore, having a basic understanding of coding can help individuals make informed decisions about technology and its impact on society.

30) List 3 or more of the different sort algorithms mentioned in the video

The different sort algorithms mentioned in the video are:

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Merge Sort
5. Quick Sort

31) What is the 'Decision Problem'?

The "Decision Problem" is a fundamental question in computer science that asks whether a computer algorithm can solve any problem that can be solved by a human. Specifically, it asks whether there exists an algorithm that can take any input and decide whether a given statement is true or false. This problem was first formulated by mathematician David Hilbert in 1928 and was later shown to be equivalent to the halting problem.

32) An example of an abstraction used in the video is, "A transistor is a type of ___"?

An example of an abstraction used in the video is, "A transistor is a type of switch."

33) Which video was the most interesting or your favorite?

My favorite and most interesting topic are The Art of Abstraction – Computerphile. Because it discusses the fundamental concept of abstraction and how it is used in computer science to solve

complex problems. Understanding abstraction is crucial for anyone interested in programming and computer science, as it enables developers to break down complex problems into smaller, more manageable parts. This video provides a clear and concise explanation of the concept and its importance, making it a valuable resource for both beginners and experienced developers.