* 1. **Describe what an abstract machine is.**

An abstract machine is the concept of having the physical hardware of a computer, separated from what the computer can do. Therefore, an abstract machine is defined by what language it can understand and what language it can run programs from, it uses an interpreter to do the conversion. It is important to understand that all physical machines have an abstract machine associated with them but not necessarily the other way around.

* 1. **Describe the functioning of an Interpreter for a generic Abstract machine.**

The role of the interpreter in an abstract machine is to fetch-decode-execute cycle. Taking in instructions from the memory and storing the result from the execution.

* 1. **Describe at least two advantages of the usage of an Interpreter in comparison with a Compiler.**

Running and testing code is easier when using an interpreter since the program can be executed step by step and there is no need for compiling. Using an interpreter also gives greater flexibility and portability

* 1. **Describe the main disadvantage of the usage of an Interpreter in comparison with a Compiler.**

The main disadvantage of using an interpreter instead of a compiler is speed. This is because the compiler can analyze the structure of a program and is allowed to rearrange the instructions to optimize the program.

* 1. **Describe what an activation record is.**

An activation record is the structure used to store a block environment in the stack. It contains information about the locally declared variables and it has a dynamic chain pointer, pointing to the latest beforehand used environment according to execution. In some implementations it contains a static chain pointer which points to the hierarchically parent block according to the structure of the code. This means that if block B is written inside block A, then the static chain pointer will point to block A.

* 1. **In the context of Memory Management, describe what the Heap is.**

The heap is a section of memory in which memory can be allocated. This allows variables and objects to exist beyond the block in which they were created. Because if they only lived in the activation record, they would be destroyed as soon as the block returned. All memory which is dynamically allocated is stored in the heap.

* 1. **Describe in what conditions the Stack is not enough to implement an interpreter or compiler for a language. Explain why.**

An activation record cannot become bigger after it has been created therefore if a block wishes to store some data with size dependent on some external input, for example a user typing a string, it cannot be stored in the activation record and requires a heap. This is called dynamic memory allocation and it cannot be done with the stack.

* 1. **Describe the main advantage that the buddy system or the Fibonacci heap are offering as heap management techniques.**

The main goal of the buddy system and the Fibonacci is to eliminate external fragmentation in the heap. They do so because they handle the heap as a tree and then based on the size of a request will take the smallest, big enough to hold the object, leaf to store a given object. Therefore, no space is left between leaves in the heap thus no external fragmentation.

* 1. **Describe what a type is.**

A type is a collection of data with a given set of operations. An example of a type is the primitive type int, this type has some operations defined for example +. A primitive type is the “building block” types which is not built on other types, and a composite type is a type built from combining other types.

Programs use types to know exactly how much memory must be allocated to store a variable of that type. A programmer can also define their own types to optimize organizing their programs. Types also help make it easy to check correctness at compile-time since it is defined what types can interact with each other.

* 1. **When are two types equivalent if the language is using structural equivalence?**

Two types are structurally equivalent if they have the same number of fields with the same types in the same order. Depending on the implementation the names of the fields matter.

The set of attributes must be the same and the attributes must be pairwise equivalent.

* 1. **Does Haskell have structural equivalence?**

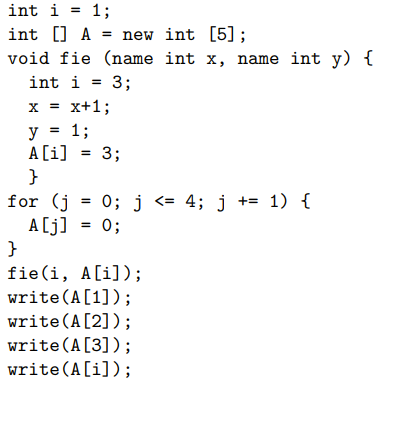
Yes, Haskell has structural equivalence.

Function types have to use structural equivalence because the type in a function definition is anonymous (the function name that appears in the definition has this anonymous type), there is no name to compare.

* 1. **Describe the goal of the unification algorithm used for performing the type inference of an expression.**

The unification algorithm can infer the type of an expression to check whether or not a type can assigned at all. If no type can be assigned an error has been made and the expression cannot be computed.

* 1. **Consider the following pseudo code of a language using static scoping and passing parameters by name. What does it print? Motivate your answer.**



In the given code snippet, the program will print the following 0 1 3 1. This is because the program starts by initializing all the integers in the array A to 0. Then

according to the book’s definition of call by name the actual arguments is executed in the environment of the caller. Therefore, x = x +1 will update the value of the i on line 1 till the value 2. Then y = 1 will become A[i] = 1 where i is referring to the i at line 1 thus updating A[2] till 1. Then A[i] = 3 will reference the local i and this will produce A[3] = 3. When printing A[i] is again referencing the i on line 1.

When the program makes the call to fie and passes the parameters by name, that means that the parameters are evaluated only when and if they are referenced by the program. This means that when it calls x = x + 1 then the global variable i is incremented by one, because the values passed are evaluated in the environment of the caller. This also means that when y is evaluated and y is A[i] we will get A[2] (because we incremented the global i ). So A[2] is then set to 1 and then A[i] is evaluated by fie, meaning in its local environment, in which it has defined its own i variable to be 3, meaning that A[3] is set to 3. Therefore when we print A[1] through A[3] we get 0, (because it was never changed) 1, (because the global i used to evaluate A[i] was incremented from 1 to 2) and 3, (because we used the local i, which was 3, to evaluate A[i]). Finally, we print A[i] from the global scope, in which we still have incremented the global i to be 2. Therefore we print A[2] which is 1. Therefore, we print 0131.

* 1. **What does the previous program print if dynamic scoping is used instead of static scoping?**

If dynamic scoping is used it will print the following 0 1 3 1.

This is the same as in exercise 4.a and the program doesn’t change because all the i’s references will be same. The last write A[i] will still reference the i on line 1 because it is in the local environment of that write.

Because the passed variables are still being evaluated in the scope of the caller, this would print the same. There might have been a difference if the caller didn’t itself define i.

* 1. **What does the previous program print if static scoping and call by value is used?**

If call by value is used instead, upon entering will be and will be . This means that in the body of the lines and have no consequence on the program since these variables are never used again. As such on the last line of , is set to three. No other positions of are changed and neither is the global . As such the program will write.

0

0

3

0

* 1. **What happens instead if static scoping and call by reference is used?**

If instead call by reference is used and are references to and ( is evaluated to before entering since static scoping typically uses deep binding)

This means that on line the global is incremented by one making it .

Similarly, line sets to . Then once again the last line sets to .

Thus, when has returned the global is and is .

Then the program writes:

1

0

3

0

* 1. **Write a function in Haskell using pattern matching that given a list of at least two integers returns 1 if the first element is 0, the second element of the list otherwise. Write also the type signature of the function.**

If the first element in the list matches the constant in the first definition returns 1. Otherwise, the second definition is used. Since it is guaranteed that there are at least two integers int the list will not throw an exception.

func :: [int] -> int

func (0:\_:\_) = 1

func (\_:y:\_) = y

* 1. **Rewrite the previous function using guards.**

myFunction :: [Int] -> Int

myFunction (x:y:xs)

| x == 0 = 1

| otherwise = y

* 1. **Describe the purpose of the keyword let in Haskell. Define a simple expression in which the let keyword is used.**

is used to assign names to constants or functions making it easy to use multiple times in the same function, essentially used the same way the keyword is.

however, is used inside a block in the same way is but when is not a monad.

This simple Haskell function uses let to assign the name “string” to the string “Hello” before printing the string

So the keyword “let” is accompanied by another keyword “in”, these 2 keywords are providing the same utility as the “where”-clause (which won’t be described into detail).

“let” simply allows the programmer to define some variables before they are (possibly) used after the “in” keyword.

Since Haskell is lazy, the definitions inside the “let” isn’t executed before they are needed/used in the “in” clause, which makes for efficient programming, that doesn’t execute unnecessary stuff.

It also makes code more readable and maintainable, since you are able to separate operations instead of making one long line of code.

Example: myLet :: Int -> Int -> Int

myLet a b = let x = a \* -1

in if a >= 0 then a + b else x + b

In the example it is shown that the x is defined in the “let”, and makes it much easier to read the final statement.

Basically the code tells you that if a is positive we add it to b, but if it is negative, we use the definition of x inside the “let” and convert the negative number to its corresponding numerical value as a positive integer.

twoTimesPi = let pi = 3.14159 in 2\*pi

1. **Briefly describe the data type you used to represent a configuration in your Onitama project and how you check that the player cards and the coordinates of the students pawns satisfy the lexicographic order.**

Estimate if the solution makes sense

1. **Write a Haskell function commonPoints that given a non-empty list of QuadTrees returns a QuadTrees of boolean pixels such that it has a pixel at True if in the same position all the QuadTrees of the list have the same pixels, False otherwise.**

To create the function commonPoints i create four functions, the commonPoints function as starting point, a runner function to check all combinations in the given list of quadtrees, a function called eqTree to create the bool QT for two given quadtrees, and a function toBool which takes a quadtree and makes all leaves contain True.

commonPoints :: [QT a] -> QT Bool

commonPoints [x] = toBool x

commonPoints (x:xs) = eqTree (runner x xs) (commonPoints xs)

runner :: QT a -> [QT a] -> QT Bool

runner y [x] = eqTree y x

runner y (x:xs) = eqTree (eqTree y x) (runner y xs)

eqTree :: QT a -> QT a -> QT Bool

eqTree (C a) (C b) = if a == b then (C True) else (C False)

eqTree (Q a b c d) (C e) = Q (eqTree a (C e)) (eqTree b (C e)) (eqTree c (C e)) (eqTree d (C e))

eqTree (C e) (Q a b c d) = Q (eqTree (C e) a) (eqTree (C e) b) (eqTree (C e) c) (eqTree (C e) d)

eqTree (Q a b c d) (Q e f g h) = Q (eqTree a e) (eqTree b f) (eqTree c g) (eqTree d h)

toBool :: QT a -> QT Bool

toBool (C a) = C True

toBool (Q a b c d) = Q (toBool a) (toBool b) (toBool c) (toBool d)

*commonPoints :: Eq a => [QT a] -> QT bool*

*commonPoints qts = commonPointsRec qts (C True)*

*commonPointsRec :: Eq a => [QT a] -> QT bool -> QT bool*

*commonPointsRec (x:[]) qtb = qtb*

*commonPointsRec (x:xs) qtb = let newqtb = QTeq x (head xs) qtb in commonPointsRec xs newqtb*

*QTeq :: Eq a => QT a -> QT a-> QT bool -> QT bool*

*QTeq (C a) (C b) (C eq) = if (a == b && eq) then (C True) else (C False)*

*QTeq (C a) (Q b c d e) (C eq) = Q (QTeq a b eq) (QTeq a c eq) (QTeq a d eq) (QTeq a e eq)*

*QTeq (Q a b c d) (C e) (C eq) = Q (QTeq a e eq) (QTeq b e eq) (QTeq c e eq) (QTeq d e eq)*

*QTeq (Q a b c d) (Q e f g h) (C eq) = Q (Qeq a e eq) (QTeq b f eq) (QTeq c g eq) (QTeq d h eq)*

*QTeq (C a) (C b) (Q i j k l) = Q (QTeq a b i) (QTeq a b j) (QTeq a b k) (QTeq a b l)*

*QTeq (C a) (Q b c d e) (Q i j k l) = Q (QTeq a b i) (QTeq a c j) (QTeq a d k) (QTeq a e l)*

*QTeq (Q a b c d) (C e) (Q i j k l) = Q (QTeq a e i) (QTeq b e j) (QTeq c e k) (QTeq d e l)*

*QTeq (Q a b c d) (Q e f g h) (Q i j k l) = Q (QTeq a e i) (QTeq b f j) (QTeq c g k) (QTeq d h l)*

* 1. **Give the type signature of main and explain what main does when executed.**

The type signature of main will be:

main :: IO ()

The code snippet of main will first take the String “” and wrap it in an IO monad. Then it will take the String out of the IO String from which it gets from getLine which takes a String from standard in and wraps it in a IO monad, using bind and store it in line, Then it will print the String to standard out, and return an empty IO monad.

* 1. **In the previous program, unfold the do syntactic sugar notation.**

Is equal to:

When the syntactic sugar is unfolded main will become the following:

main = return “” >> (getline >>= (\line -> putStrLn line))

This can be unfolded another time giving:

main = return “” >>= (\\_ -> getline >>= (\line -> putStrLn line))

* 1. **Show the steps that Haskell takes to compute the following expression.**

Text

Description automatically generated

* 1. **Define the type signature of the two expressions. What will we obtain by evaluating the two expressions?**

Ex1 = *map (\x -> x) [[0,1],[2]]*

Ex2 = *mapM (\x -> x) [[0,1],[2]]*

Ex1 :: [[Integer]]

Ex2 :: [[Integer]]

Ex1 = [[0,1],[2]]

Ex2 = [[0,2],[1,2]]