Brandon Chester

Ada Maze Solver

Program design process and strategy

For my Ada project, I chose to implement a program that can solve mazes using a stack. The stack approach essentially works the same way as the recursive method for solving mazes, since recursion is really creating an implicit stack. My approach to writing the program actually took inspiration from my experience with the first assignment. What I realized is that it can be difficult to implement complex code within a language that you're still learning and are generally not familiar with. Therefore, I felt it would be best to work with a newer language that I do know, and that already offers many of the structures I would need to implement myself in Ada.
I've attached my full Swift maze-solving program along with this submission. Here I will go over the basic steps I went through when implementing the solution there, and my work to port it back to Ada.

```
struct Stack<Element> {
    var items = [Element]()

    mutating func push(_ item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
}
```

My first step was to implement a Stack. The Swift code for my stack structure is above. This is actually a good example of why I chose Swift as my implementation language. Having arrays be mutable and resizable by default made it very easy to wrap an array type inside a struct to create a stack class with the standard push and pop operations that would be required for the program.

```
class Position {
    let previous:Position?
    let x:Int
    let y:Int

    init(_ x: Int, _ y:Int, _ p:Position?) {
        self.previous = p
        self.x = x
        self.y = y
    }
}
```

The code for my stack in Swift stack takes in a generic element type which could be an instance any Swift structure or class, but the Ada implementation is specific to the Position class that I defined. Above you can see the Swift version of this class, which is essentially the same as the Ada version. The x and y variables correspond to the

position in the maze, and so x is the column in the 2D array, while y is the row, and both begin at 0 to keep things aligned with Swift array indices. The previous field holds a reference to the previous position in the maze, which was necessary to go backward along the path to find the true path from start to finish without the dead-end branches.

```
func solveMaze(_ grid:[[Character]]) {
    var maze = grid
    var curr = Position(0, 0, nil)

    for row in 0..<maze.count {
        for column in 0..<maze[row].count {
            if maze[row][column] == "o" {
                curr = Position(row, column, nil)
            }
        }
    }

    // Do DFS using Stack
    var stack = Stack<Position>()
    stack.push(curr)
    while stack.items.count != 0 {
        curr = stack.pop()
        print("X: \(curr.x). y:\(curr.y)")
        if maze[curr.x][curr.y] == "e" {
            stack.items.removeAll()

            // Go back through the previous positions
            var solvedMaze = grid
            var temp:Position! = curr.previous
            while (temp.previous != nil) {
                solvedMaze[temp.x][temp.y] = "@"
                temp = temp.previous
            }

            for row in 0..<solvedMaze.count {
                print(String(solvedMaze[row]))
            }
        } else if maze[curr.x][curr.y] != "+" {
            maze[curr.x][curr.y] = "*"
            stack.push(Position(curr.x, curr.y + 1, curr))
            stack.push(Position(curr.x, curr.y - 1, curr))
            stack.push(Position(curr.x - 1, curr.y, curr))
            stack.push(Position(curr.x + 1, curr.y, curr))
        }
    }
```

Finally, we get to the actual implementation. This is essentially just a translation of the depth-first search pseudocode provided with the assignment itself. The only change occurs when the solution is found. The program actually keeps two copies of the maze read from the file. The first is modified while the algorithm runs, with squares being marked as they are traversed. However, when the tile representing the end is found, the program then traverses through the previous positions. Put simply, when a position object is created, it stores a reference to the position that the program was currently on when it moved there. This allows it to traverse backward through the positions that make up the true path from the start to the end of the maze, which are then marked in

the second copy of the maze array so the maze can be printed without the dead-end paths.

As far as the Ada implementation goes, I originally thought that it would be a direct port of the Swift code, but I had to make a few changes that I didn't expect. For example, due to my need for records that reference other records of the same type, and an unknown number of items, I ended up implementing a stack type which uses a linked list internally, rather than the array that I used in the Swift code. This required creating a node class which would wrap the position objects in the list, and I also had to create classes called NodePtr and PositionPtr which could reference their respective types. The stack itself is also implemented in a separate package from the main program, while in the Swift program all code exists in the same file.

Beyond that, the differences in the code mainly relate to how Ada differs from Swift syntactically and structurally, and both programs are fairly similar in their core implementation. The Ada program also has some defensive-programming aspects, such as ensuring that the file name provided by the user is valid, and that the file follows the correct format. The Ada program will also output a message and then exit if a maze is not connected, which makes it unsolvable.

Greatest problems faced

I encountered a number of issues when trying to implement this program in Ada, even though I easily implemented my Swift version to use as a basis. Much of this stems from the design of Ada as a language.

My first issue was that Ada is simply too verbose. Jean Ichbiah, the creator of Ada, stated "C was designed to be written; Ada was designed to be read." The problem is that he clearly misunderstood what makes something readable. He incorrectly conflated having an English-like syntax with being readable. In fact, Ada has some of the worst readability I've ever seen in a programming language, being even worse than Objective-C which also fell into the same trap of equating verbosity and English syntax with readability.

Human beings are able to learn languages with varying syntaxes. What really matters is having a consistent and concise grammar. Adding the verbosity of English to a programming language just makes the code bloated and repetitive. For example, it's not necessary to have if statements enclosed in a "then, end if" section. Human beings are able to understand the meaning of the brace-brackets used in other languages, even though they don't represent actual English words. This trend continues in other areas, like writing "procedure procedure_name is, begin, end procedure_name" to wrap a procedure. Having both "is" and "begin" is completely redundant, and the verbosity of the declaration doesn't make it any more clear than brace-brackets.

The syntactical issues lead to my second problem, which was the compiler. Ada's compiler is amazing at catching errors at compile time, but the verbosity of the

language means that you're always fighting to make your code compile because you forgot some sort of decoration word that needs to be added in the code. I don't think I ever successfully changed an Ada file and had it still compile after the changes were saved.

As for implementation issues, the largest one I had was how Ada handles reference types. I originally tried to implement my stack using an array of the position types that I created. However, having a type in Ada that references a record of the same type is not as easy as it is in other programming languages. In a language like C, explicit pointers allow the easy implementation of structs that self-reference their type. In object-oriented languages, reference semantics mean that self-referencing types is not an issue as all.

In Ada, you need to define a separate type which access the associated type, which I described in the section above when discussing my PositionPtr and NodePtr types. I agree that reference and value types should be differentiated, which is something that languages like Java fail to do. However, Ada's implementation is very clumsy and makes it difficult to work with both forms of a record type, and I found this shocking given that Ada is an object-oriented language.

What makes Ada a good language?

I think including argument labels for parameters passed to a subroutine is a great feature of Ada and other languages. I do feel that Ada's syntax for it is not as good as other languages, but the inclusion of the feature is something to be commended, and the structure of it allows you to pass parameters out of order so long as the correct labels are attached, which isn't possible in a language like Swift or Objective-C.

Having actual multidimensional arrays is also a nice language feature. Most other languages resort to having arrays of arrays, which can be complicated when you scale to several dimensions where you no longer have some geometric interpretation of the layout of the elements in memory.

Another good feature of Ada is the way it performs for loop using a range syntax. The C-style for-loop is potentially one of the worst programming language design decisions to ever be made, as it's a huge source of off-by-one errors and other confusion relating to the scope of the counter variable. Ada's "for x in q..r" syntax is explicit, easy to read, and much harder to get wrong than a C-style for-loop.

The last feature that stands out as a good aspect of Ada is the use of variable intent labels for procedure. It's always helpful to know whether a parameter is going to be modified by a procedure, and given that Ada differentiates functions from procedures, it's essential in being able to pass data back from a procedure. In other languages with reference-semantics, it's difficult to know whether a method will modify what you pass to it unless stated in documentation, and Ada avoids this by explicitly putting the rules related to the use of the variables within the declaration.

Ease of implementation compared to C

Implementing this program in C would be much easier than it was in Ada. For one, not having to deal with different forms for using types, and variables that access those types, would cut down on the complexity of implementation significantly. In C I could simply define a stack struct, which internally contains a linked list of nodes containing position structs, and a size counter. All structs would be allocated dynamically, and there wouldn't be any need for separate types that access structures, as pointers are explicit and accessible in C. Not having to cluster all variable declarations outside the implementation code would also make the program more compact, and C's syntax would allow for more concise and actually more readable code than the horrendously verbose Ada code.

Difficulty of learning Ada

As I mentioned earlier, Ada has a number of language features that aren't really favoured in modern programming languages. This made it much harder to learn than other languages, including Fortran which I had to learn for the previous assignment. This is not necessarily simply due to Ada being different than other languages, but more so in the fact that design of Ada is not very pragmatic despite its intentions. You constantly feel like you're fighting with the language just to do basic things, and getting code to compile is a struggle in its own right.

Greatest dislike about Ada

I already commented on this in the section about challenges faced, but it's worth restating that the thing I dislike most about Ada is how it conflates verbosity with readability. It is actually much harder to read than a language like Swift, and it is predicated on the notion that making programming statements read more like English will make them easier for people to read. This misunderstands what actually makes something readable for human beings, and I think it's why Ada has largely been shunned and forgotten as a programming language, and why almost none of its practices exist in modern programming languages today. I would rather be unemployed than work in a capacity where I have to develop programs using Ada.