

MAZE SOLVER

A NONDETERMINISTIC COMPUTING APPROACH TO MAZE SOLVING



December 15, 2015

**Markous Soliman & Moustafa Elshaabiny**

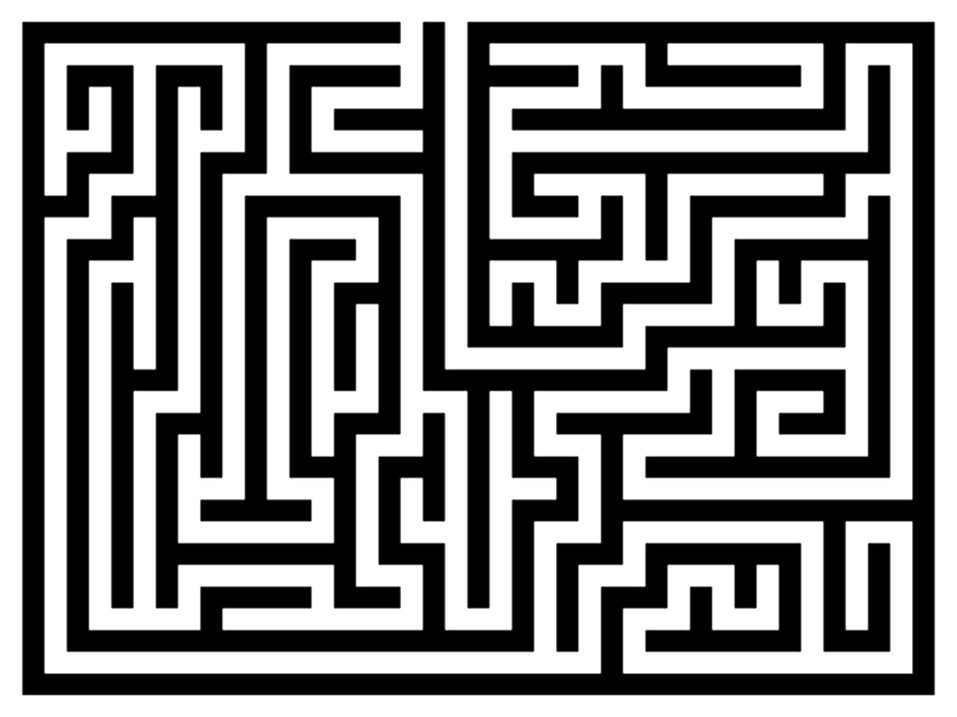


Table of Contents:

1. Introduction
2. Project Scope
   1. Nondeterministic computing
      1. Maze problem
3. Materials and Methods
   1. *amb* operator
   2. Continuations and Continuation passing
4. Maze Solver
5. Discussion
6. Conclusion
7. Recommendations
8. **Introduction:**

This report is intended to provide a brief overview of functional programming and its core concepts, a high-level explanation of nondeterministic computing, and, finally, to showcase an interesting real-world application to demonstrate the effectiveness of the concepts mentioned above. In our report, we assume a basic understanding of programming and software design terminology. However, our intention is to shine a light onto the functional programming arena in a reader-friendly manner.

Although it can be daunting, functional programming is one of the most powerful and respected programming paradigms nowadays. In fact, it is considered one of the key players in the “big data” revolution[[1]](#footnote-1). Essentially, functional programming is about adopting a programming style and a set of ideas that break down code into smaller pieces, which can be easily debugged and reused. The two main advantages that distinguish functional programming from other programming styles are immutability and statelessness. Simply put, immutability allows software engineers to write cleaner code, better abstractions, and, more interestingly, concurrent programs. Immutable is not the same thing as unchangeable, however. In functional programming, we create new data structures to store the changed data rather than overwriting the data in an existing one. On the other hand, the stateless nature of functional programs prevents any previous knowledge from interfering with the programs execution. In other words, stateless programs execute every task as if it is the first time. This nature allows them to operate in a ‘vacuum' without relying on outside values to do their calculations, and a function call can have no effect other than to compute its results. In order to perform tasks, those programs only use the parameters being passed to them (later on, we illustrate how we utilized this feature in our Maze Solver).

Those features mentioned earlier motivate the idea of distributed systems. With the current trends shifting to computing with more, rather than faster, processors, giant companies like, Google, Microsoft, IBM, and Facebook, now run their programs in data centers. Very soon, most companies will shift to cloud computing. Thus, developing software for these distributed (also known as cloud) environments becomes essential and learning functional programming becomes a must.

Over the course of this semester, our team developed a love-hate relationship with functional programming, mainly due to the simple fact that it challenged every assumption we had about writing software.

1. **Project Scope:**

When we chose nondeterministic computing as our project area, we immediately thought of being in lost in a maze and presented by several options not knowing which one leads to the correct path to solving it. Think about this for a second. What would you do? What would a Scheme program do? Well, we designed a nondeterministic computing system that find the correct path to solving any maze in a couple of seconds. We will demonstrate how exactly does the system do that in subsequent sections of this report.

The essence of nondeterministic computing is an automatic search that is invaluable to “generate and test” types of applications. The maze problem is one such application of that type. Moreover, we use bounded non-determinism, where every process is confined to a finite number of choices (You should not be stuck in a maze with an infinite number of directions you can take!). In the next section, we describe some of the methods and operations that the nondeterministic approach utilizes.

**Methods and Operations:**

**3.1- *Amb evaluator***

The idea of *amb* is difficult to understand, at first, but here we will try to give a high-level description of what it is and simplify things a little bit. Amb is a nondeterministic operator that takes zero or more expressions and makes an “ambiguous” (or nondeterministic) choice among them. Those expressions represent the possible values of the amb expression. Amb’s choice does not guarantee successfully returning a value that is accepted by our program, however. Amb is always restricted by a set of rules that makes the results of certain choices invalid. In this case, amb uses a sleek backtracking mechanism (called call-with-current-continuation) to try other possible values until it finds one that returns a valid result. Should amb’s choice cause final failure, the program will backtrack to the chronologically previous amb call. In a sense, the restrictions that we impose on amb in our programs veto amb’s choice to what we want it to have picked from the beginning.

There are two types of non-determinism: *weak and strong non-determinism[[2]](#footnote-2).* Amb falls under the weak non-determinism umbrella[[3]](#footnote-3), where its choice can follow a certain order and is not “fully” nondeterministic. In other words, we should expect amb to choose the first expression’s value first followed by the second, and so on. As long as it fails our assertions, amb will try the next possible choice. Eventually, it will make a sequence of correct choices that ultimately returns the desired results. The backtracking happens behind the scenes. As a result, it might seem like amb is always making the correct choices. It might be tempting to think that amb has super powers, but we are here to tell you that everyone makes mistakes, even amb! The only difference is that amb “knows” where it made that mistake and “travels” back to the point in time and tries a different choice.

***3.2- Continuation Passing***

Current continuation and continuation passing plays a vital role in amb, so it is important to explain what they mean and how they work to fully understand amb. “During the evaluation of a Scheme expression, the implementation must keep track of two things: (1) what to evaluate and (2) what to do with the value”[[4]](#footnote-4). Here is an example in pseudo-code, assume we want to evaluate the following expression: if x is *true* do y, otherwise do z. “What to evaluate” is if x is *true* and and "what to do with the value" is to make the decision which of y or z to do/evaluate *and* to do so. In functional programming (and Scheme in specific) the “what to do with the value” is called the *continuation* of a computation. An legal Scheme expression has a continuation ready to continue the computation for that point.

1. https://ascent.atos.net/the-rise-of-functional-programming/ [↑](#footnote-ref-1)
2. http://comjnl.oxfordjournals.org/content/35/5/514.full.pdf [↑](#footnote-ref-2)
3. we used the macro implementation of amb [↑](#footnote-ref-3)
4. http://www.scheme.com/tspl3/further.html [↑](#footnote-ref-4)