The City College of New York

CSC 33500 Programming Language Paradigms

Spring 2016

Professor Douglas Troeger

A Nondeterministic Computing Approach to NFA

Group Members: Aaron Bridgemohan & Weifan Lin

May 27, 2016

Table of Contents

1. Introduction

The purpose of this project report is to provide a brief overview of functional programing and a detailed explanation of nondeterministic computing, and to showcase an interesting application which demonstrates the way that backtracking can be used to solve a nondeterministic finite automata. In this paper we will present some background information on how nondeterministic computing works, how backtracking works and its benefits, and also how a nondeterministic finite automata functions. We will also give a brief explanation why backtracking is a useful solution to the finite automata. The goal of our project is to implement a scheme language system that performs pattern matching on a nondeterministic finite automata.

2. Background

2.1. Nondeterministic Computing

Nondeterministic computing, like stream processing, is useful for “generate and test” applications. The key idea is that expressions in a nondeterministic language can have more than one possible value. A nondeterministic program evaluator will work automatically choosing a possible value and keeping track of the choice. If a subsequent requirement is not met, the evaluator will try a different choice, and it will keep trying new choices until the evaluation succeeds, or until we run out of choices. Just as the lazy evaluator freed the programmer from the details of how values are delayed and forced, the nondeterministic program evaluator will free the programmer from the details of how choices are made.

It is instructive to contrast the different images of time evoked by nondeterministic evaluation and stream processing. Stream processing uses lazy evaluation to decouple the time when the stream of possible answers is assembled from the time when the actual stream elements are produced. The evaluator supports the illusion that all the possible answers are laid out before us in a timeless sequence. With nondeterministic evaluation, an expression represents the exploration of a set of possible worlds, each determined by a set of choices. Some of the possible worlds lead to dead ends, while others have useful values. The nondeterministic program evaluator supports the illusion that time branches, and that our programs have different possible execution histories. When we reach a dead end, we can revisit a previous choice point and proceed along a different branch.

2.1.1. Nondeterministic Computing Example

The following snapshot of a program starts with two lists of positive integers. The goal is to find a pair of integers, one from the first list and one from the second list whose sum is prime. This is considered to be a fully nondeterministic program because it would begin to check the sum of an element from the first list and an element from the second list and keep iterating until the solution is prime.

```
(define (prime-sum-pair list1 list2)
  (let ((a (an-element-of list1))
        (b (an-element-of list2)))
    (require (prime? (+ a b)))
    (list a b)))
```

2.2. Backtracking

There are functions such as reversing a file or binary search where each recursive call only makes one call. This type of call is also known as the tree of calls where the calls form a linear line from the initial call to the base case. The performance of such an algorithm is dependent on how deep the function stack gets. The issue with this type of recursion is that there is multiplicative factors that are carried down a tree. By this I mean that a tree of function calls has multiple branches at each level, which in turn has multiple branches of it own and that continues to the base case. With linear recursion this can become very complex. Let's say for example purposes, you were looking for a specific base case on a tree, this is where backtracking would come in handy. Backtracking will allow your function to work through the tree to an initial base case and if that is not the base case you were looking for then it will go back to a decision point and try and alternative. This beats you having to go back to the beginning and trying over and over again to find the case you were looking for. If your function ends up backtracking all the way to the initial state and all alternatives have been explored then you can come to the conclusion that there is no solution for your problem

2.2.1. Backtracking Example

As an example, we will show how backtracking works to find the good base case which is the (E) leaf in this tree. The first thing backtracking would do is start at the root, and choose one of the two options which is A and B. Let's start with A. At A, the next options would be C and D. You go to C and it is bad so here is where backtracking comes into play and goes back to A. Remember at A you had two options so instead of going all the way back to root you would check D. You will find that D is bad so you backtrack to A. Turns out A has no more options so you backtrack to root and move forward to B because that is the other option. From B you have two options which is E and F. You move to E and found that that is a good leaf. This is where backtracking would stop and output that E is your solution or has been found.

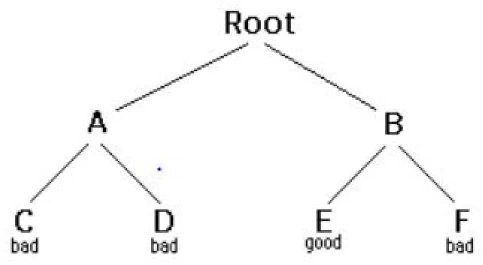

Figure 1: Tree

2.3. NFA (Nondeterministic Finite Automata)

NFA is represented formally by a 5-tuple, (Q, Σ, Δ, q, F). Q represents a finite set of states, Σ represents a finite set of input symbols, Δ represents a transition function, q represents an initial (or start) state, and F represents a set of states that is the accepting state or final state. The first condition of an NFA is that the machine must start at the initial state q. The second condition states that for each character of the given string w, the machine will transition from state to state according to the transition function Δ. The last condition states that the machine should accept string w if the last input of the string w causes the machine to halt in one of the accepting state F. If the machine is unable to get from the start state to an accepting state by following w, then the machine will reject the string.

2.3.1. NFA Example

As a small example we will use the NFA creates below to explain how the string is accepted. We can see that there is a start state to the left ($q\varepsilon$) and a final state (qp) to the right. Let us say we have a input string of w = {1 0 0 1}. The machine will check the first string input, 1, at the start state and loop back to the start state accepting that first string. It will then move on to the second string, 0 and see that zero is a transition from start state to state q0. After it makes that transition and accepts that second string input, it will go to the third string, 0 and once again transition to state q00. It will do the same for the last string input 1, and then come to a halt in the accepting state which means that NFA machine accepts all the input strings. If however that last string, 1 was not in the complete input string then the machine would have halted at q00 and therefore not accept the input strings.

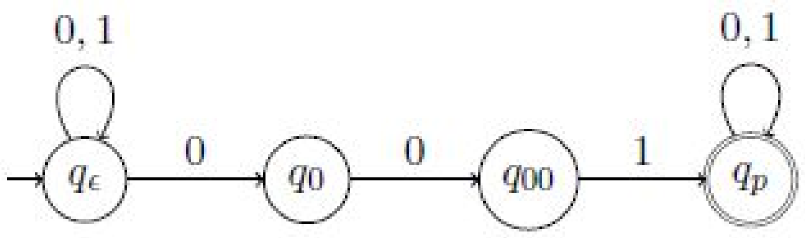

Figure 2: NFA Example

In addition, a transition can be an epsilon transition with a epsilon symbol, Ɛ, as shown in the figure below. With an epsilon transition, anything input can be accepted.

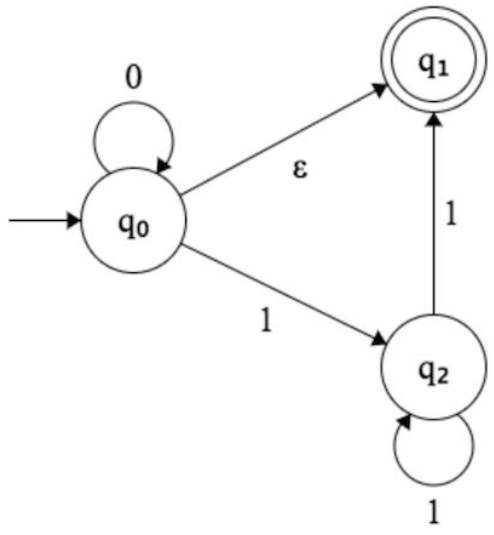

Figure 3: NFA with Epsilon Transition

3. <u>Details of Implementation</u>

Our program is a scheme interpreter which processes strings as directed by an NFA and returns whether or not that strings can be accepted by the NFA . Firstly, we will need to create an NFA. As we mentioned earlier, an NFA is represented by a finite set of states, a finite set of input symbols, a transition function, a start state, and a set of accepting states. So our NFA is represented as the following form:

'((accepting states) (state (symbol transitions)))

For example, (define m1 '((4) (1 (a 1 2) (b 1) (c 1)) (2 (a 2) (b 3)) (3 (c 4)))), and the figure below shows what m1 NFA looks like.

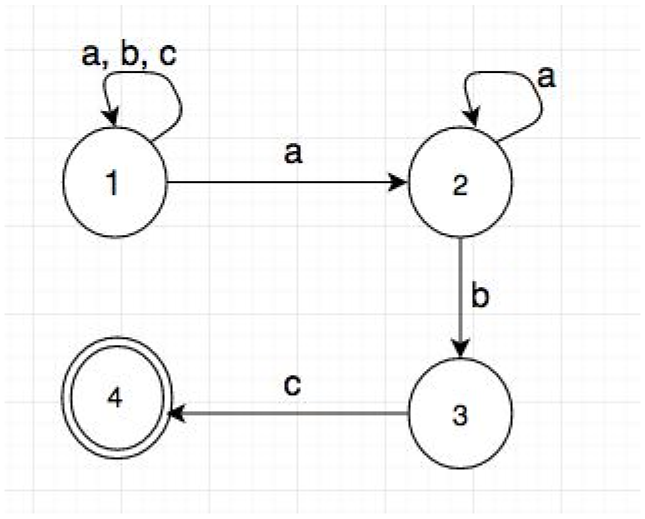

Figure 4: Finite State Machine m1

When we want to execute NFA, then we can specify the start state. Next, we need to find out the next states for a given state and a symbol. For the example above, the next states for state 1 would be state 1 and state 2 if the input symbol is a. To do that, we need the following functions.

```
(define (sub-machine state Machine)
  (cond
    ((null? Machine) '())
    ((member? state (car Machine)) (cdar Machine))
    (else (sub-machine state (cdr Machine)))
    )
  )
```

sub-machine takes a state and a machine, returns a list of the transitions from that state. For the same example, if we want to get the sub-machine of m1 from state 1, (sub-machine 1 m1) would return ((a 1 2) (b 1) (c 1)) which is shown in the following figure.

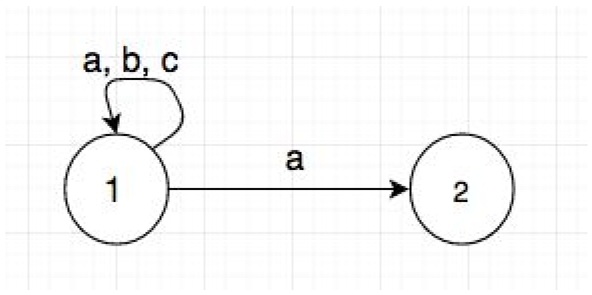

Figure 5: Sub-machine From State 1

```
(define (next-states symbol Machine)
  (cond
    ((null? Machine) '())
    ((equal? symbol (caar Machine)) (cdr (car Machine)))
    (else (next-states symbol (cdr Machine)))
    )
)
```

next-states takes a symbol and a machine (sub-machine), returns a list of possible destination states given the symbol. For example, the sub-machine in the Figure 5, if the given symbol is a, the result list would be (1, 2), and if the symbol is c, the result would be just (1).

```
(define (transitions state symbol Machine)
  (cond
    ((null? Machine) '())
    (else
     (next-states symbol (sub-machine state (cdr Machine))))
    )
  )
```

transitions takes a state, a symbol and a machine, returns a list of possible destination states from the input state via the input symbol.

Keeping track of the correct state is the most important procedure in our program, as it tells the program on which moves are valid and invalid that require backtracking. The following function does the backtracking for our program. The main purpose of this function is to run through every possible path for a given input string, after the last character of the string has been read and if the current state is an accepting state, then the singleton list of that current state will be returned.

```
(define (backtracking string start next-states eps-states Machine)
  (cond
    ((and (null? string) (not (atom-member? start (end-state Machine)))) '())
    ((and (null? string) (atom-member? start (end-state Machine))) (list start))
    ((and (null? next-states) (null? eps-states)) '())
    ((null? next-states)

     (if (null? (nfa-execute string (car eps-states) Machine))
         (backtracking string start next-states (cdr eps-states) Machine)
         (cons start (nfa-execute string (car eps-states) Machine))))

    (else
     (if (null? (nfa-execute (cdr string) (car next-states) Machine))
         (backtracking string start (cdr next-states) eps-states Machine)
         (cons start (nfa-execute (cdr string) (car next-states) Machine))))
    )
  )
```

Finally, we have our last function, for executing NFA which takes an input string, a start state, and an NFA, returns the list of transition states if the input string can be accepted by the NFA.

```
(define (nfa-execute string start Machine)
  (cond
    ((null? string)
     (if (not (member? start (end-state Machine)))
         '()
         (list start)))
    (else (backtracking string start (transitions start (car string) Machine) (transitions start 'eps
Machine) Machine))
    )
  )
```

4. Results

For testing, we created the following NFAs, m1 and m2 as shown in the Figure x.

```
(define m1 '((4) (1 (a 1 2) (b 1) (c 1)) (2 (b 3)) (3 (c 4))))

(define m2 '((3) (0 (a 1) (eps 2)) (1 (b 3)) (2 (a 2 3))))
```

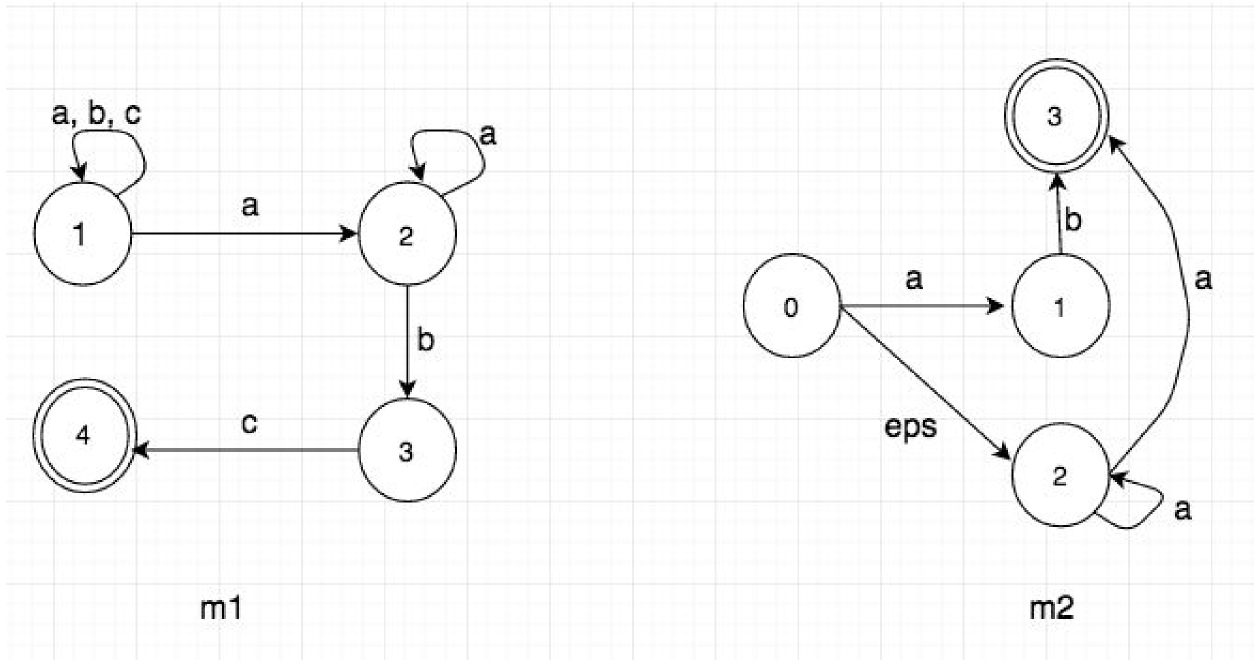

Figure 6: m1 & m2 NFAs

Several tests were tested on these two NFAs to assure the achievement of the goal:

- Input string: '(a b c), start state: 1, NFA: m1. Result is '(1 2 3 4).
- Input string: '(b a c b a a b c), start state: 1, NFA: m1. Result is '(1 1 1 1 1 1 2 3 4).
- Input string: '(a b c c), start state: 1, NFA: m1. Result is '().
- Input string '(a b), start state: 0, NFA: m2. Result is '(0 1 3).
- Input string: '(a a a a a a a a a a a), start state: 0, NFA: m2. Result is '(0 2 2 2 2 2 2 2 2 2 2 2 3)
- Input string '(a), start state: 0, NFA: m2. Result is '(0 2 3)

Based on these tests, our program seems to give correct results.

5. Conclusion

As evidenced through the results, the goal of the project was successfully accomplished. We managed to implement a scheme language system that performs pattern matching on a nondeterministic finite automata. Before this project we were not proficient on nondeterministic machines, or backtracking, however we now understand the basic concepts of how they can be useful as well as implemented. We found that backtracking is a useful method because it allows you to exhaust all options before returning a no solution. Without backtracking a program would take longer and be less efficient to find an accepting state because it would keep iterating from the beginning each time is checks a list or tree branch. We also found that functional programming is a very powerful tool when trying to implement backtracking on a nondeterministic finite automata. Learning functional programming and applying it to a finite automata has been a fun experience and really pushed us to understanding its usefulness.

6. References

Zabiht, R., & McAllester, D., & Chapman, D. (1987) *Non-Deterministic Lisp with Dependency-Directed Backtracking*. Retrieved from http://www.aaai.org/Papers/AAAI/1987/AAAI87-011.pdf

Bacchus, F. *A Uniform View of Backtracking*. Retrieved from http://www.cs.toronto.edu/~fbacchus/Papers/uniform-backtracking.pdf

7. Appendix

```scheme
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Project Scope: Non-deterministic Computing
; CSC 33500 Programming Language Paradigms
; Spring 2016
; Weifan Lin & Aaron Bridgemohan
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



; member?. This function checks if an atom is in a list.
(define (member? a li)
  (cond
    ((null? li) #f)
    (else (or (equal? a (car li))
              (member? a (cdr li))))
    )
  )



; sub-machine. This function takes a state and a machine, returns a list of the transitions from that state.
(define (sub-machine state Machine)
  (cond
    ((null? Machine) '())
    ((member? state (car Machine)) (cdar Machine))
    (else (sub-machine state (cdr Machine)))
    )
  )
```

; next-states. This function takes a symbol and a machine (sub-machine), returns a list of possible destination states

```
; given the symbol.
(define (next-states symbol Machine)
  (cond
    ((null? Machine) '())
    ((equal? symbol (caar Machine)) (cdr (car Machine)))
    (else (next-states symbol (cdr Machine)))
    )
)
```

; transitions. This function takes a state, a symbol and a machine, returns a list of possible destination states from

```
; the input state via the input symbol.
(define (transitions state symbol Machine)
  (cond
        ((null? Machine) '())
        (else
         (next-states symbol (sub-machine state (cdr Machine))))
        )
      )
```

```
    ; end-state. This function returns an accepting state of a machine.
    (define end-state
      (lambda (machine)
        (car machine)))
```

```
    ; backtracking. This function is the most important procedure in our program, as it tells
the program on which moves are
    ; valid and invalid that require backtracking. The following function does the backtracking
for our program. The main
    ; purpose of this function is to run through every possible path for a given input string,
after the last character of
    ; the string has been read and if the current state is an accepting state, then the
singleton list of that current state
    ; will be returned.
```

```
(define (backtracking string start next-states eps-states Machine)
  (cond
    ((and (null? string) (not (atom-member? start (end-state Machine)))) '()) ; string is null & start != end => empty list
    ((and (null? string) (atom-member? start (end-state Machine))) (list start)) ; string is null & start = end => end
    ((and (null? next-states) (null? eps-states)) '())  ; next-states and eps-states are null => empty list
    ((null? next-states) ; next-states is null, check eps-states

     (if (null? (nfa-execute string (car eps-states) Machine))
         (backtracking string start next-states (cdr eps-states) Machine)
         (cons start (nfa-execute string (car eps-states) Machine))))

    (else ; eps-states is null, check next-states
     (if (null? (nfa-execute (cdr string) (car next-states) Machine))
         (backtracking string start (cdr next-states) eps-states Machine)
         (cons start (nfa-execute (cdr string) (car next-states) Machine))))
    )
  )



; nfa-execute. This function takes an input string, a start state, and an NFA, returns the list of transition states
; if the input string can be accepted by the NFA.
(define (nfa-execute string start Machine)
  (cond
    ((null? string)
     (if (not (member? start (end-state Machine)))
         '()
         (list start)))
    (else (backtracking string start (transitions start (car string) Machine) (transitions start
'eps Machine) Machine))
    )
  )
```

```
; Testing
(define m1 '((4) (1 (a 1 2) (b 1) (c 1)) (2 (b 3)) (3 (c 4))))
(define m2 '((3) (0 (a 1) (eps 2)) (1 (b 3)) (2 (a 2 3))))



(nfa-execute '(a b c) 1 m1) ;returns '(1 2 3 4)
(nfa-execute '(b a c b a a b c) 1 m1) ;returns '(1 1 1 1 1 1 2 3 4)
(nfa-execute '(a b c c) 1 m1) ;returns '()
```

```
(nfa-execute '(a a a a a a a a a a a) 0 m2) ;returns ’(0 2 2 2 2 2 2 2 2 2 2 2 3)

(nfa-execute '(a b) 0 m2) ;returns '(0 1 3)

(nfa-execute '(a b a a a b) 0 m2) ;returns '()

(nfa-execute '(a) 0 m2) ;returns ’(0 2 3)

(nfa-execute '() 0 m2) ; returns '()
```